

COSC1082-Computer Organisation

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Chapter 1

Data Representation

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe and use number systems in different bases
- Convert numbers between binary, decimal hexadecimal and octal
- Perform additions and subtractions in various numbers systems

Digital Information

Information is very often represented and managed in a digital format: photos, pictures, video, music, data are translated to 0s and 1s, and manipulated that way:

- a character can be represented $A = 01000001$,
 $B = 01000010 \dots$
- colours on a scalar scale can be represented by
 $\text{white} = 000000 \dots \text{black} = 111111$
- yes/no or true/false can be represented by $\text{yes} = 1$,
 $\text{no} = 0$, etc.
- an open switch may be a 1, a closed switch may be a 0

The meaning of the sequence of 0s and 1s is a matter for interpretation: the symbol 01000001 may be interpreted as the character A or the number 65.

Digital Information

In this course we are interested in the architecture and functioning of general purpose digital computers:

- From the software/programmer point of view, all information inside a computer are 0s and 1s
- From the engineering point of view these are two voltages, 0 volts is interpreted as a 0, +5 volts is interpreted as a 1

Positional Systems

- Our arithmetic is based on the 'positional system' where the position of a digit within a number determines its actual value; the number 2 in 123 means twenty, but the number 2 in 2,345 means two thousands
- The positional value of each column, starting from the right is 1, 10, 100, 1000, etc. That is $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, etc.
- The decimal number system has a Base or Radix of ten, using ten different digits or symbols in the system 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
- All integers are made up of a combination of these digits, in the positional system described before

Positional Systems

- A number system can be based on any radix such as 2, 5, 8, 10, 16, 27. The arithmetic rules and the addition and multiplication rules are quite the same in any system. For example, if we work to the base 5, we count 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, ...
- The addition rules look different to the decimal ones:

$$1 + 2 = 3$$

$$2 + 3 = 10$$

$$3 + 3 = 11$$

$$4 + 3 = 12$$

- The times tables also change:

$$2 * 1 = 2$$

$$2 * 2 = 4$$

$$2 * 3 = 11$$

$$2 * 4 = 13$$

Positional Systems

What would be the 3 times table to the base 5? (Complete the table)

Table: 3 multiplication table, to the base 5

Operation	Result
$3 * 1$	
$3 * 2$	
$3 * 3$	
$3 * 4$	

Positional Systems

We can also perform other operations that are similar to the standard decimal ones. As an exercise, complete the operations in the table (only one has been completed):

Table: Some operations to the base 5

Operation	Result
3^2	14
2^3	
4^2	
3^3	

Important Number Systems

The most important number systems for digital computers are:

- binary: base 2. Digits are: 0, 1
- octal: base 8, Digits are: 0, 1, 2, 3, 4, 5, 6, 7
- hexadecimal: base 16. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

We shall see that hexadecimal and octal numbers are very useful when dealing with binary numbers, and we'll learn to use and do arithmetic in each of them

Binary Numbers

- The basis for binary numbers is 2
- Radix = 2
- The symbols are: 0,1

The positional value of each binary digit indicates the power of 2, as follows:

Table: A binary number

digit	1	1	0	0	1	1	0	1
power	128	64	32	16	8	4	2	1
exponent	7	6	5	4	3	2	1	0

Binary Numbers

- According to the previous table the number 11001101 is equivalent to decimal:
$$1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 128 + 64 + 8 + 4 + 1 = 205$$
- **Exercise:** find the decimal equivalent of the binary numbers:
1101 =
1111111 =
- **Note:** a binary digit is called a bit. The bit on the right-hand side is the Least Significant Bit (LSB), while the bit on the left-hand side is the Most Significant Bit (MSB)
- Digits may have meaning for different bases: 10010 which may be a binary, octal, decimal or hexadecimal number. To indicate the basis, use 10010_2 , 74562_8 and 74562_{10} .
- Sometimes a prefix is used, as in % for binary and \$ for hexadecimal.

Counting in Binary

Decimal	Binary	Decimal	Binary
00	00000	11	
01	00001	12	
02	00010	13	
03	00011	14	
04		15	
05		16	
06		17	
07		18	
08		19	
09		20	
10		21	

Exercise: complete the table.

Hexadecimal Numbers

- Binary numbers are very easy for computers, but not for human beings
- Compare 1010111100110011111000000101 with the equivalent hex AF33D05
- Hexadecimal and octal are important because of the ease of conversions between both systems and binary.
- The basis for hexadecimal numbers is 16
- Radix = 16 (2^4)
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Hexadecimal Numbers

Table: Hexadecimal Numbers

Positional values:	16^4	16^3	16^2	16^1	16^0
Decimal values:	65,536	4096	256	16	1

Therefore the number:

$$\begin{aligned} 1D7E_{16} &= 1 * 16^3 + D * 16^2 + 7 * 16^1 + E * 16^0 \\ &= 1 * 4096 + 13 * 256 + 7 * 16 + 14 * 1 = 7550_{10} \end{aligned}$$

Exercises: Convert to decimal

\$7E =

\$A8 =

\$13BF =

Counting in Hexadecimal

Decimal	Hexadecimal	Decimal	Hexadecimal
00	00	10	
01	01	11	
02	02	12	
03	03	13	
04		14	
05		15	
06		16	
07		17	
08		18	
09		19	

Exercise: complete the table. Use prefix \$ or subscript 16, as in \$1FED or 1FED₁₆

Octal Numbers

- The basis for octal numbers is 8
- Radix = 8 (2^3)
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7

Table: Octal Numbers

Positional values:	8^4	8^3	8^2	8^1	8^0
Decimal values:	4096	512	64	8	1

$$\begin{aligned} 1573_8 &= 1 * 8^3 + 5 * 8^2 + 7 * 8^1 + 3 * 8^0 = 1 * 512 + 5 * 64 + 7 * 8 + 3 * 1 \\ &= 512 + 320 + 56 + 3 = 891_{10} \end{aligned}$$

Counting in Octal

Decimal	Octal	Decimal	Octal
00	000	11	
01	001	12	
02	002	13	
03	003	14	
04		15	
05		16	
06		17	
07		18	
08		19	
09		20	
10		21	

Exercise: complete the table

Binary to Decimal

Using the positional value of each digit, we can calculate the decimal expression of a binary number:

Table: Binary to Decimal

2^4	2^3	2^2	2^1	2^0
16_{10}	8_{10}	4_{10}	2_{10}	1_{10}

For example:

$$10101_2 = 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1 = 16 + 0 + 4 + 0 + 1 = 21_{10}$$

Decimal to Binary

Repeatedly divide by 2; for example, convert 28_{10} to binary

2)	28	
2)	14	0
2)	7	0
2)	3	1
2)	1	1
2)	0	1

In the algorithm above, the resulting binary number should be read upside down.

Therefore: $28_{10} = 11100_2$

Decimal to Binary

The above algorithm can be explained as follows. According to the above sequence, we can write:

$$28 = 14 * 2 + 0$$

$$14 = 7 * 2 + 0$$

$$7 = 3 * 2 + 1$$

$$3 = 1 * 2 + 1$$

$$1 = 0 * 2 + 1$$

Then,

$$\begin{aligned} 28_{10} &= 14 * 2 + 0 = (7 * 2 + 0) * 2 + 0 = 7 * 2^2 + 0 * 2^1 + 0 * 2^0 = \\ &= (3 * 2 + 1) * 2^2 + 0 * 2^1 + 0 * 2^0 = 3 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = \\ &= (1 * 2 + 1) * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = \\ &= 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 11100_2 \end{aligned}$$

Exercises

- ① Convert the following decimal numbers to binary:

8

17

32

64

127

128

255

- ② What is the largest number that may be represented using 4 bits? 8 bits? 16 bits? and 32 bits?
- ③ How many bits do you need to represent 1000_{10} ? and 100000_{10} ? and 100000000_{10} ?

Hexadecimal to Binary

- Hexadecimals are more convenient for human beings, and it is very easy to convert between them and binary.
- To convert hexadecimal to binary, write each hexadecimal digit as a 4-bit binary number and put it all together as in the following example:

Hexadecimal	F	D	6	9	A
Binary	1111	1101	0110	1001	1010

- The binary expression of \$FD69A is %11111101011010011010. In particular, since a byte is 8 bits, each memory location may be represented by 2 hexadecimal digits.

Binary to Hexadecimal

From binary to hexadecimal we break down the binary number into groups of 4 binary bits (padding with 0s to the left if necessary), as in:

Table: Binary to Hexadecimal

Binary	0101	1110	1010	1011	1111	0001
Hexadecimal	5	E	A	B	F	1

NOTE: Start at the right hand end and fill the left hand end with zeros to get 4 bits if necessary.

Between octal and binary

These conversion are the same as the hexadecimal conversions, but instead of dividing into groups of 4 you have to divide into groups of 3. For example:

$$111\ 001\ 101\ 100_2 = 7154_8$$

Decimal Addition

To add in decimal, we learn some rules that are appropriate for that number system, such as $2 + 3 = 5$ and $7 + 6 = 13$. Using these rules we have learnt how to add multi-digit numbers such as:

$$\begin{array}{r} 56 \\ + 97 \\ \hline 153 \end{array}$$

When we add $6 + 7$ we obtain 13, so we write down a sum of 3 and a carry of 10 to the next column:

$$\begin{aligned} 56 + 97 &= 50 + 6 + 90 + 7 = 50 + 90 + 6 + 7 = 50 + 90 + 13 \\ &= 50 + 90 + 10 + 3 = 150 + 3 = 100 + 50 + 3 = 153 \end{aligned}$$

Binary Addition

The rules are the same as for decimals, but remembering that $1 + 1 = 10$, so when adding $1 + 1$ you have to write down 0 and carry 1.

	Sum	Carry
$0 + 0 =$	0	0
$0 + 1 =$	1	0
$1 + 0 =$	1	0
$1 + 1 =$	0	1

Example:

$$\begin{array}{rcccccc}
 & 1 & 0 & 1 & 0 & = 10_{10} \\
 + & 0_0 & 0_1 & 1_0 & 1 & = 03_{10} \\
 \hline
 0 & 1 & 1 & 0 & 1 & = 13_{10}
 \end{array}$$

Exercises

Perform the following binary additions:

$$1010 + 0011 =$$

$$1110 + 0001 =$$

$$1101 + 0001 =$$

$$1101 + 1001 =$$

$$1110 + 1101 =$$

$$0101 + 1000 =$$

$$0001 + 1001 =$$

Data Sizes

- Computers deal with data in a fixed number of bits, 8, 16, 32, 64 bits
- Errors may occur when these sizes are exceeded
- For example in $1001\ 1101 + 1000\ 1001 = 1\ 0010\ 0110$ the leftmost bit (in bold) of the result is a *carry out* bit
- The result of the operation does not fit into 8 bits
- If we don't consider the carry out bit, this operation is (in decimal):
 $157 + 137 = 38$ which is clearly wrong
- The carry out bit is telling us that the result exceeds the 8 bits capacity, and that the result is not valid

NOTE: For unsigned data (more on this later), if the carry out bit is 1 the result is invalid

Decimal Subtraction

Consider the operations:

$$\begin{array}{r} 7 \\ - 4 \\ \hline +3 \end{array} \qquad \begin{array}{r} 73 \\ -17 \\ \hline 56 \end{array}$$

The left hand side presents no problems

On the right hand column of the second example we have $3-7$ which we can't do. So, we add the base 10 to the 3 to give 13 producing a borrow of one to the next column (that is, an actual borrow of 10), and then subtract the 7 to give a difference of 6

Decimal Subtraction

What we are actually doing is:

$$\begin{array}{rclclcl} 73 & = & 60 & + & 13 & & \\ & & - 10 & - & 7 & = & \\ & = & 50 & + & 6 & = & 56 \end{array}$$

Binary Subtraction

Similarly, we can subtract binary numbers — remembering that all the operations are binary — like so:

$$\begin{array}{rcccccl} & & 1 & & & \\ & 1 & 0 & 1 & 1 & = 11_{10} \\ - & 0_1 & 1 & 1 & 0 & = 6_{10} \\ \hline & 0 & 1 & 0 & 1 & = 5_{10} \end{array}$$

Table: Binary Subtraction Rules Table

	Difference	Borrow
$0 - 0 =$	0	0
$0 - 1 =$	1	1
$1 - 0 =$	1	0
$1 - 1 =$	0	0

Binary Subtraction

NOTE:

- The top number is larger than the bottom number. Since we are at the moment dealing with only positive numbers (unsigned), it is not possible to subtract a larger number from a smaller one. If we do, the result will be invalid
- The need to borrow is indicated by the single 1 at the top, and the little 1 on the right-hand side of the last column
- A borrow out of the last column indicates that the top number is smaller than the bottom number, and so the result is invalid

Hexadecimal Addition

- By memorising the addition of hexadecimal digits, we can add hexadecimals. Similarly to base 10, this is the 'add tables' in the base 16.

$3 + 5 =$	8	$5 + 5 =$	A
$3 + 6 =$	9	$5 + 8 =$	D
$3 + 7 =$	A	$7 + 8 =$	
$3 + 8 =$	B	$8 + 8 =$	
$A + 3 =$		$A + 4 =$	
$A + 5 =$		$A + 6 =$	
$B + 7 =$		$D + 8 =$	

- We have to memorise hexadecimal addition tables
- Exercise:** complete the table

Hexadecimal Addition

Example:

$$\begin{array}{r} \$1F4C3 \\ + \$729AB \\ \hline \$91E6E \end{array}$$

$$3_{16} + B_{16} = E_{16} \text{ and carry} = 0$$

$$C_{16} + A_{16} = 16_{16} \text{ sum } 6_{16} \text{ and carry} = 1$$

Hexadecimal Multiplication Table

We can also construct multiplication (times) tables:

Table: Some hex operations

$2 * 3 = 6$	$2 * 8 = 10$	$2 * D =$
$2 * 4 = 8$	$2 * 9 = 12$	$2 * E =$
$2 * 5 = A$	$2 * A = 14$	$2 * F =$
$2 * 6 = C$	$2 * B =$	$3 * 8 =$
$2 * 7 = E$	$2 * C =$	$3 * A =$

Hexadecimal subtraction

- By remembering the addition rules of hexadecimal digits, subtraction is straightforward. Consider the following example:

$$\begin{array}{r}
 \text{\$F}_1 \quad 5_1 \quad \text{B} \\
 - \quad \text{\$3} \quad \text{A} \quad 9 \\
 \hline
 \text{\$B} \quad \text{B} \quad 2
 \end{array}$$

- When the top hex digit is too small, we do the same as in decimal arithmetic. For example, for the final column we have:

$$\text{B}_{16} - 9_{16} = 2_{16} \text{ and borrow} = 0$$

$$5_{16} - \text{A}_{16} = \text{B}_{16} \text{ and borrow} = 1$$

$$\text{F}_{16} - 3_{16} - 1_{16} \text{ (the borrow)} = \text{B}_{16}$$

Binary Numbers Range

Table: N° of bits vs. Range

Bits	Range/Max Value	N° of values
8	0 - 255	256
16	0 - 65,335	65,536
32	0 - 4,294,967,295	$2^{32}-1$
64	0 - 18,446,744,073,709,551,615	$2^{64}-1$

The maximum value is 2^n-1 , i.e. for 8 bits it is $2^8-1 = 255$

Chapter 2

Data Representation-Basic Computer Organisation

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe how negative numbers may be represented in a computer system
- Describe the 2's complement number system
- Perform operations that include negative numbers using the 2's complement representation
- Describe the main components and operation of a general purpose computer and their operation, including CPU, memory and buses

Negative Binaries

Using 0s and 1s we have been able to represent positive numbers. To represent negative numbers the two most important systems are:

- Signed Magnitude: The most significant bit is used to indicate the sign, using 1 = negative and 0 = positive. For example, binary 0000 0110 is used to represent +6, and binary 1000 0110 is used to represent -6
- Two's Complement: also the MSB is 1, but the negatives start at the other end (more on this later)

Signed Magnitude

- The MSB is used, 1 = negative and 0 = positive, the rest are the same
- It is clear and easy to implement
- However, the operations do not conform to the standard arithmetic rules, as in:

$$\begin{array}{rcl} & 0000 & 0110 & = + 6 \\ + & 1000 & 0110 & = - 6 \\ \hline & 1000 & 1100 & = - 12 \end{array}$$

- Bad for circuitry, because all the circuits are designed to carry out the rules of standard arithmetic

2's Complement

- The choice of negatives in sign magnitude is a poor one from the point of view of the operations
- We should select the numbers so that half of the numbers are positive and the other half are negative, but in a different way
- Considering 8 bits, we choose the negatives starting at the other end of the interval:

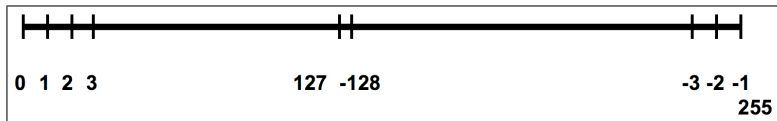


Figure: 2's complement representation

2's Complement

- When we add in 2's complement two opposite numbers we get the result 1 0000 0000 (i.e. 256)
- If we ignore the carry out we get all zeros, which is the result we want
- This is more convenient from the point of view of the operations:

$$\begin{array}{r} 1 = \quad 0000\ 0001 \\ - 1 = \quad 1111\ 1111 \\ \hline 1\ 0000\ 0000 \end{array}$$

2's Complement

Summarising:

- unsigned numbers, no negatives, 0 to 255
- 2's complement: 0 to 127 positive, -128 to -1 (255) negative
- If we ignore the carry out we can use the existing electronic circuits
- In a processor, the operation is performed, the 0000 0000 byte is left in the register the *carry bit C* is set to 1 if there is a carry out (Note: some processors set more than one bit)
- Programmers can check these bit/s to find out whether there was a carry. If we only consider 8 bits the operation is correct

Positional in 2's complement

- An alternative way of looking at the 2's complement representation. Consider for an 8-bit number:

Binary	-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$	$+2^1$	$+2^0$
Decimal	-128	64	32	16	8	4	2	1

- In this representation, the number:

$$1111\ 1111 =$$

$$-1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 =$$

$$-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1$$

NOTE:

- The most significant bit has a negative contribution
- In 8 bits the binary unsigned sum of a number and its 2's complement is: 1 0000 0000 = 256
- = 0 in 8-bits 2's complement

Interpretation of Operations

- When operating with negative values, we have to be careful how we interpret the operands and the results of the operations. For example, the following sum:

$$01000110 + 10101100 = 11110010$$

may be interpreted as an unsigned operation, or as a signed 2's complement operation:

	0100 0110	=	70	=	70
+	1010 1100	=	172	=	-84
	<hr/> 1111 0010	=	242	=	-14

- NOTE:** The operation *is the same*, only the *interpretations* are different. Both operations may be implemented with the same circuits. Further, the processor will simply perform the operation and set or reset some bits according to the result

Getting the 2's Complement of a Number

- The rules for conversions are based on flipping the bits of the binary number by changing 0s to 1s and 1s to 0s — this is called the 1's complement — and adding binary 1 to obtain the 2's complement expression of a number x
- The complete algorithm is as follows (for 8 bits):

```
if 0 <= x <128
```

```
    the 2's complement expression is the same as  
    the original number
```

```
else
```

1. Find the 1's complement of x by flipping the bits
2. Add 1
3. The result is the 2's complement expression of x

2's Complement Expression

Example: find the 2's complement expression of -35_{10}

$$35_{10} = 0010\ 0011_2$$

1. flip each of the bits above	1101 1100
2. add 1	+ 0000 0001
3. result	1101 1101

2's Complement Addition

- In 2's complement addition there are also invalid results if the sum falls outside the acceptable range of numbers
- In unsigned addition an invalid result is detected by a carry out of the MSB; in this case the 68K sets the carry out bit $C = 1$ and the extend bit $X = 1$: the result is invalid for unsigned data
- In 2's complement operations, the overflow is detected by the *overflow bit V*. If $V = 1$, the result is invalid.
- We have to look at the carry into the last column and the carry out of the last column. If both carry bits are the same (both 0 or both 1), the overflow bit $V = 0$; if they are different, the overflow bit $V = 1$

2's Complement Addition

Example:

$$\begin{array}{r}
 \\
 + 1_0 1_0 1_0 1_0 1_0 0 \\
 \hline
 1 0 1 1 1 1
 \end{array}$$

In the example, the carry into the last column is 0, and the carry out is 1; therefore the carries are different, $V = 1$ and the result is invalid

Control Bits

- In an unsigned subtraction the top number must be larger than the bottom number
- The processor performs the operations, and sets/resets some bits according to the result
- The program/programmer must determine what to do with the result

In the Motorola 68000:

- If the operation produces a borrow, the carry C and the extend X bits are set (=1), otherwise they are reset (=0)
- If the result is negative, the negative bit N is set, otherwise it is reset
- If the result is zero, the zero bit Z is set otherwise it is reset
- if the operation results in overflow, the overflow bit V is set, otherwise it is reset

Control Bits

- This allows the programmer to determine exactly the fate of the operation
- Following our previous example, for the sum:

$$\begin{array}{r} 0100\ 0110 \\ +\ 1010\ 1100 \\ \hline 1111\ 0010 \end{array}$$

- Both the C and V bits are reset to zero, so the result is valid as a signed and unsigned operation
- The result may be interpreted as valid either as the sum of the two unsigned numbers, or as the sum of two signed numbers
- Question: what are these two values?

Control Bits

- Now the operation:

$$\begin{array}{r} 1100\ 0110 \\ +\ 1010\ 1100 \\ \hline 1\ 0111\ 0010 \end{array}$$

results in a carry bit $C = 1$ and a V bit $= 1$ (since the carry in is zero and the carry out is 1), and so both results are invalid

Exercise:

Show an example of an operation in which $C = 0$ and $V = 1$, and another one in which $C = 1$ and $V = 0$

Subtraction with 2's Complement

- With 2's complement we can subtract a number from another and get a correct result, provided the numbers are within the 2's complement range
- We calculate the opposite of the number in the 2's complement notation, and add
- If we want to do $0001\ 1110 - 0110\ 1100$ we first have to find the expression of the opposite of the subtrahend $0110\ 1100$:

flip the bits		1001 0011
add 1	+	0000 0001
result		1001 0100

- Now perform the addition with the opposite just calculated:

	0001 1110
+	1001 0100
	1011 0010

- Please note that the result is the number -78_{10} , that is, a negative result

Subtraction with 2's Complement

- Each binary number in the following subtraction may represent either a 2's complement value or an unsigned value:

binary	2's comp	unsigned
11011110	-34_{10}	222_{10}
-11010011	$-(-45)_{10}$	-211_1
<hr/>		
00001011	$+11_{10}$	11_{10}

- The computer does not care whether the data is unsigned or 2's complement
- It is up to the user to interpret the operation and check the C or V bits
- In the example above $C = 0$ and $V = 0$, so the results are valid in all interpretations

Subtraction: Signed vs. Unsigned

This approach to subtracting two 2's complement binary numbers *cannot* be used for *unsigned* binary numbers.

For unsigned binary numbers, it is *necessary* to perform the subtraction as you would for two decimal numbers.

In summary:

- if the numbers are unsigned, perform the subtraction as for two decimal numbers
- if the numbers are signed, add the opposite of the subtrahend

Common Confusion

- We have to be careful how we talk about numbers and the 2's complement format
- Consider carefully the following statements:
 - 1 Find the 2's complement expression of -35_{10} : this is $1101\ 1101_2$, as we have seen before
 - 2 Find the number that is the opposite of 35_{10} in the 2's complement format: this is -35_{10} , with the expression as above
 - 3 Given a number x , $-x$ is the number that is the opposite of x in the 2's complement format, e.g. -35_{10} is the 2's complement of 35_{10} , 18_{10} is the 2's complement of -18_{10}
 - 4 When we are talking of operations in the 2's complement format, to calculate $x - y$ we should:
 - 1 calculate the number $-y$ (opposite of y in the 2's complement format), and
 - 2 add this to x

Computer Operation

- The CPU (Central Processing Unit) operates by fetching and decoding each instruction, and then executing the instruction.
- The CPU contains an *Arithmetic and Logic Unit (ALU)* and a few registers with a fixed number of bits 16, 32 or 64 bits
- The CPU transfers data into its — ALU and registers — where operations are performed, and then writes the results back to memory

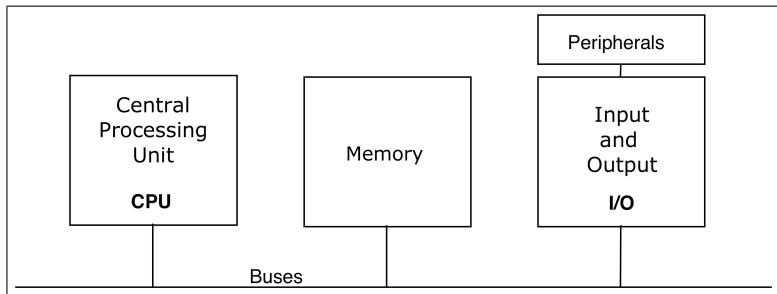


Figure: A computer block diagram

Memory

Memory:

- stores the data and instructions of currently running programs (each is called a *process*)
- Main memory stores instructions and data in locations numbered sequentially: 0, 1, 2, 3, 4, 5,
- Each memory location is called an *address*
- Individual bits are not addressable, memory is *byte addressable*
- N bits can express up to 2^N binary numbers, so the size of memory is limited by the number of bits used in an address
- The largest memory that may be addressed with 16 bits is 2^{16} , 0 ... $2^{16}-1$

Memory Locations

The diagram illustrates a memory structure with eight rows. On the left, a bracket labeled 'addresses' groups the address values. On the right, a bracket labeled 'contents' groups the binary values. Arrows point from the 'contents' label to the binary strings in the second, fourth, and sixth rows.

000008	11001101
000009	11001000
00000A	10111000
00000B	10000001
00000C	00001000
00000D	11001111
00000E	10101000
00000F	11001001

Figure: Contents of Memory locations

CPU

CPU:

- Fetches decodes and executes instructions stored in memory
- It includes the ALU and registers
- Other spaces such as
 - MAR (Memory Address Register)
 - MBR (Memory Buffer Register)
 - PC (Program Counter)

I/O System

The I/O system:

- Connects the computer to external peripherals such as disks, keyboard, monitor, mouse, printer and network
- The CPU communicates with some interface registers to handle the data transfers
- This may be implemented by making the I/O addresses part of the memory, *memory-mapped I/O*, or
- Using special instructions to access special I/O registers, *instruction-based I/O* (also called *programming-based I/O*)

Buses

Buses:

- A bus is a collection of wires carrying binary information between subsystems: the CPU, registers, memory and I/O devices
- A bus is normally shared between several devices, so there must be some sort of agreement on how to use the bus: *bus protocol*
- A bus may be connecting two specific devices, *point-to-point bus*, or it can be a common pathway for several devices, called a *multi-point bus*

Process Memory Allocation

- Programmers write programs in a high level language
- A compiler translates this to machine instructions
- The compiler works with the operating system to allocate memory for the program to run; this is dependent on the system but a common one is:

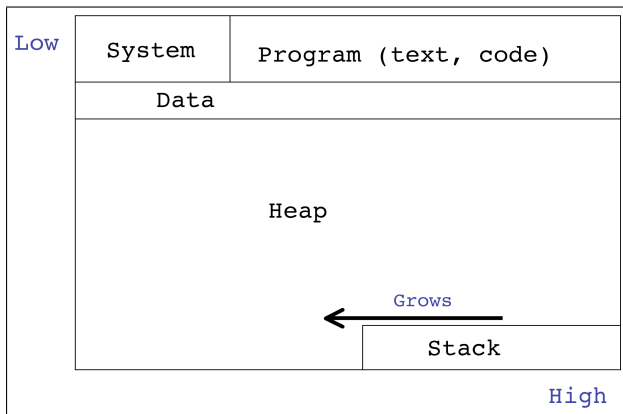


Figure: Typical memory configuration

Clocks

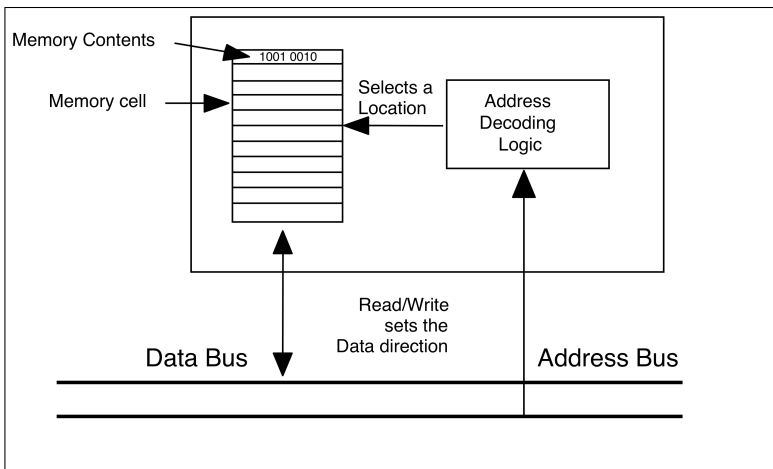
- The operations of a computer are synchronised by a computer clock
- This is a hardware device that produces a very regular sequence of 'ticks'
- Modern desktop computers clocks can tick in excess of 3 billion times per second
- That gives an indication of the speed of the computer
- Large computers and super computers are much faster than personal computers

Basic Operation

- The number of parallel wires in the bus is called the *bus width*
- It corresponds to the number of bits allowed at one time on the bus
- A bus would have some lines dedicated to:
 - transfer data (*data bus*)
 - handle addresses (*address bus*)
 - control lines (*control bus*)
- When fetching data in, the CPU puts the memory address to be accessed on the address bus, sets the Read/Write line to read, and then the data is transferred on the data bus usually into the CPU area
- When placing data in memory, the CPU places the destination location on the address bus, sets the Read/Write line to write, and puts the data on the data bus for the data to be placed in memory

Basic Operation

- Usually, data and instructions are loaded into some of the CPU registers, operated on, and saved back. When processing an instruction, it is typical of computers to transfer words (2 bytes in the 68K) or long words (4 bytes in the 68K), instead of bytes



Chapter 3

The 68K Processor

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe the fetch, decode and execute cycle
- Discuss the main characteristics of RISC and CISC computers
- Discuss the different layers of abstraction of a computer system
- Describe the different types of memory technology
- Explain the different types of memory hierarchy, including cache and virtual memory

The Central Processing Unit

- Almost all of the operations of the computer are performed in the CPU
- Includes a set of registers where information may be stored while processing, and it has access to the buses to transfer information to and from memory

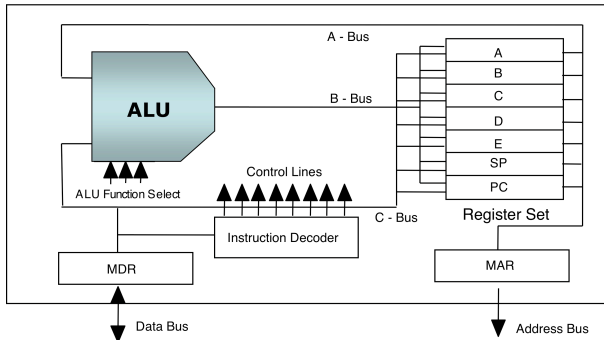


Figure: Main components of a CPU

The Central Processing Unit

The main components of the CPU are:

- ALU, Arithmetic and Logic Unit: performs the arithmetic and logic functions, including add, multiply, compare and branch
- Register Set: is for the internal storage of the current data being manipulated
- CPU Buses: Transfer data from register to register, registers to memory, registers to ALU, and ALU to registers
- MAR, Memory Access Register: stores the memory address to be accessed next via the address bus

The Central Processing Unit

- MBR, Memory Buffer Register: also known as MDR, Memory Data Register. Stores the data just read from memory or the data ready to be written to memory
- Program Counter: stores the address of the *next* instruction to be executed
- Instruction Register: holds the next instruction to be executed
- Stack Pointer: keeps the location of the top of the stack, so stack operations may be performed properly
- Instruction Decoder: Converts a program instruction into the sequence of operations that executes the instruction

An Instruction

Let's consider the instruction 'add E,B': Add the contents of register E to the contents of register B and save the result into register B

This instruction may be implemented as follows:

- ➊ Transfer the contents of register E to an ALU input via the A-Bus
- ➋ Transfer the contents of register B to the other ALU input via the C-Bus
- ➌ Select add as the ALU operation
- ➍ Transfer the result from the ALU to register B via the B-Bus

Fetch, Decode and Execute

The standard computer cycle continuously **Fetches** and then **Decodes** and **Executes** the next instruction. In each iteration,

- 1 The PC is moved to the MAR to access the memory location of the next instruction, and the CPU also sets the Read/Write line to read
- 2 The MAR drives the address into the bus, and that is translated to a memory location that is then accessed
- 3 The contents of the accessed memory location are copied onto the Data Bus and moved into the CPU's Instruction Decoder for decoding
- 4 The Instruction Decoder decodes the instruction and uses its Control Lines to prepare to execute the instruction, by clocking data into registers, enabling registers onto buses, selecting ALU functions, and so on
- 5 The ALU executes the instruction
- 6 The PC is properly incremented to the address of the next instruction, ready for the next Fetch
- 7 GOTO 1

Complex Instruction Set Computers (CISC)

- In the beginning instructions and addressing modes were minimal
- As chips improved, instruction sets and addressing modes expanded to make assembly language easier to write and supposedly more efficient
- The instruction set of CPUs became bigger, as more and more instructions were created to satisfy the needs of programmers and compilers
- Complex instruction sets produce shorter programs, and therefore use less memory, although there tends to be a proliferation of similar instructions, such as several different ways of accessing memory
- Instructions are of varying length, and some of them require many clock cycles to complete
- CPU chips become extremely complex and this took considerable chip space
- The large number of gates generated a large quantity of heat that was hard to dissipate

Complex Instruction Set Computers (CISC)

- Studying programs running in computer systems, it was found that approximately 45% of the instructions are related to data movement, 25% to ALU operations and 30% to branching and flow of control.
- Computer designers found that many of the complex instructions and addressing modes were infrequently used by compilers and users
- The size of the complex logic reduced the number of registers available for executing programs, and the different instruction lengths meant a longer, more complex decoding cycle resulting in slower execution
- There were many sub-routine calls, many memory accesses to save, return addresses and parameters and returned results
- Although some of the disadvantages may be overcome by high speed caches and wide and fast data buses (as we shall see later), the idea of producing simpler instruction sets took hold

Reduced Instruction Set Computers (RISC)

- To simplify the design and operation of the CPU, designers introduced instruction sets in which all instructions are the same size
- This allowing the interspersing of operations of different instructions — ‘pipe-lining’
- Simpler instructions resulted in most instructions able to execute in shorter clock cycles — most execute in one clock cycle — so although there were more instructions they executed much faster
- The Control Unit is much simplified, and as a consequence the silicon chip of a RISC is about a quarter the size of a CISC CPU, and generates much less heat
- With less logic, designers can use some space for more registers — 256+ — so compilers and programmers can use them for temporary storage and parameter passing, therefore saving time by accessing registers rather than memory so frequently

Computer Hierarchy

User	Executable Programs
High-Level Language	C++, Java, Ada
Assembly Language	Assembly Code
System Software	Operating System
Machine Code	Instruction Set
Control	Microcode
Digital Logic	Gates, Circuitry

Figure: Computer Hierarchy

Computer Hierarchy

- A computer operation is organised in a layered fashion
- Each lower layer offers a specific set of services to the layer above
- This relieves application programmers from the need to understand how to perform machine-dependent tasks, such as allocating memory, controlling memory access, managing disks and handling of files
- The Operating System provides many of these services to higher layers
- The OS itself relies on services from lower layers to implement its services
- This is safer
- Applications in a higher-level language are more portable, since the high-level code that runs on one machine may be (mostly) re-compiled to run on a different machine

Computer Hierarchy

- Each layer is given an Application Programming Interface (API) a set of access points or functions
- It is then possible to execute lower-level services
- For example, an application does not access the computer I/O hardware directly; that would be too dangerous
- Instead, a 'call' is made to software in the Operating System to supply the I/O service
- This approach makes these operations safer, and hides lower layer details from the upper layers, hiding differences between computers making it possible for the same high-level application to run on different hardware

Computer Hierarchy

A typical operation such as an array memory allocation in C or Java may work as follows:

- 1 an application programmer allocates an array within the code by declaring the array, say

```
int myArray[10];
```
- 2 during compilation the compiler includes a request to the OS — using a system call such as `memget()` — for contiguous space within the program execution space
- 3 when the program runs, the OS attempts to satisfy the program memory request (it may fail!)
- 4 if successful, the CPU accesses the array memory locations as required by the program
- 5 when the program runs, at all times the OS protects each area of memory against accidental overwriting, and controls that each memory access is legal

Types of Memory Technology

There are essentially two different types of core (main) memory technology:

- Static: faster memory, using Flip-Flops (we shall see Flip-Flops later). It is faster to change state between 0 and 1 — to *switch state* — and it keeps state until the power is off, but it is more expensive
- Dynamic: slower memory using capacitors. It is slower to switch, but it is possible to fit more units into a given space, it is cheaper to manufacture and it generates less heat. The problem is that capacitors lose their charge very quickly, so there is a need to refresh the contents after a few ms, otherwise they go to zero and lose the information

Memory Technology

Naturally, the faster the memory is, the faster the computer can perform its operations. However, other variables also contribute to the efficiency of the computer, such as:

- the speed of the CPU, since the faster the CPU the more operations it can perform in a given time interval
- the switching speed of memory
- the amount of memory available, because a larger portion of the running program will be available in memory ready to run (see Virtual Memory below)
- the width of the bus (to avoid multiple transfers for one memory access)
- the quality of the peripheral cards such as video and network cards, since they take load off the CPU

Memory Technology

In general, the measure of computer performance is time. The processor dependent time is given by the formula:

$$ExecTime = NumofInstructions * AveClockCyclesPInstruction * ClockCycle$$

A computer with a smaller clock cycle, fewer clock cycles per instruction or with a smaller number of instructions to do a job will have a better performance.

There are several measures of computer performance: two of the most common ones are:

- MIPS: Million of instructions per second. Since instructions are so different from machine to machine, it does not give a very accurate measurement
- MFLOPS: Megaflops, millions of floating-point operations per second. The measure gives an indication of how fast complex floating-point operations run on the processor

The Memory Hierarchy

- Some types of technology are more expensive but more efficient than others
- Faster, more efficient memory reserved for frequently used operations, and slower less efficient memory for less often used accesses
- Memory closer to the CPU is more efficient than more removed memory

Table: Memory Hierarchy and Technology

Memory Type	Size (PC)	Access Times	Technology
Registers	1K, 2K, 4K bits	1-2 ns	SRAM
Cache Level 1	8K-16K	3-10 ns	SRAM
Cache Level 2	256K-512K	20-40 ns	SRAM
Main Memory	512K-1G	30-90 ns	DRAM
Hard Disk	80G-200G	5-20 ms	DRAM

Cache Memory

- To speed up CPU instruction and data fetches from memory, processor designers often use more expensive but very fast memory called *cache memory*
- It stores a copy of the data and instructions as they are fetched by the CPU
- Thus, when the CPU executes a program loop or repeatedly accesses the same memory region the information is likely to be in the cache, which then can supply the information very quickly

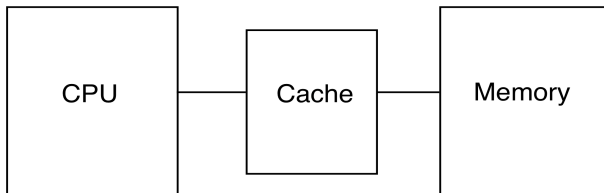


Figure: Cache memory

Virtual Memory

- For the CPU to execute an instruction or to operate on a data item they need to be somewhere in memory
- Usually a whole program does not fit in memory all at once
- Only the relevant parts of a process (a running program) are loaded into main memory, with the remainder residing on disk
- When a program requests data that is not in main memory — e.g. an instruction, or data from a file — it has to get it from disk
- This is organised transparently by the OS and the corresponding software and hardware, so programmers write programs as if the size of the computer memory is unlimited
- Since this arrangement provides programmers with a virtual computer memory to work with, it is called *virtual memory*

Virtual Memory

The access times in the Table give an indication on the relative efficiency of the different accesses.

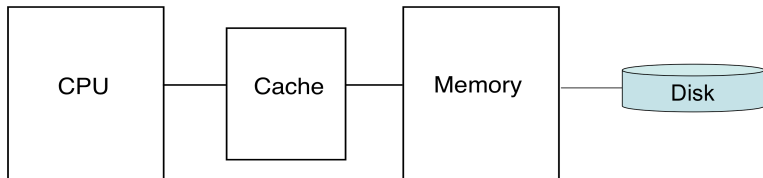


Figure: Virtual memory

Virtual Memory

To see how this works, assume a word is requested by the CPU. The word:

- is first searched for in the cache/s (primary level memory)
- if the word is in the cache (a *cache hit*), it is loaded immediately
- if it is not in the cache (a *cache miss*), it is loaded from main memory (secondary level memory) if it is there
- if it is not in main memory either, it must be brought in from disk

Virtual Memory: Paging

- With virtual memory, programs are written as if the memory capacity is only limited by the available capacity of the hard disk.
- When the required data is not in memory there is a significant performance penalty since I/O devices are several orders of magnitude slower than the processor and memory
- Having a larger memory makes the computer run faster, since the OS may keep a larger portion of the process data in memory at one given time
- The most common way to implement virtual memory is by *paging*:
 - dividing up main memory into fixed-size blocks called *page frames* usually 1K, 2K or 4K
 - dividing programs and data also into *pages* of the same size
 - when the required data is not in memory, the corresponding page where the data is in secondary memory is brought in, and placed in one of the frames, and the data is accessed there

Virtual Memory: Paging

- When the data is in memory, the running program — the *process* — waits for the transfer
- When data is required from disk however, the running process is put to sleep — it cannot run anyhow since the data it needs is not available — waiting for the transfer of the data from disk:
 - a whole page is requested to be transferred from disk
 - the incoming page is loaded into memory where it is now available for access
- Contiguous program pages do not necessarily result in contiguous page frames in memory
- Virtual memory works by translating each *logical* (*virtual*) memory address into a physical memory address.
- In this way it is possible to:
 - write programs as if there was only one user using the computer
 - write programs independently of the memory capacity of the computer

Virtual Memory: Paging

All this operation is managed by the Operating System, which is in charge of determining whether the data is in memory and, if not, of locating where the data is on disk, and:

- 1 putting the process to sleep
- 2 if there is not enough room in memory, deciding which page is to be replaced to make room for the incoming page
- 3 issuing the read transfer
- 4 loading the page into memory (this accepts variations)
- 5 waiting until the transfer has finished

When the transfer is finished, the OS decides which process is going to run in the CPU next — it may not be the original process now sleeping — and dispatches this process to run in the CPU by copying all the process information into the CPU structures (registers, PC and IP).

Chapter 4

The 68K Structure and Assembly

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe the basic structure of the 68K chip
- Discuss the stack structure and its functionality
- Describe the structure of the 68K chip
- Explain the most commonly used instruction of the 68K instruction set
- Create, assemble and run a simple assembly language program using the 68K simulator
- Explain the concept of memory alignment
- Compare the two storing strategies big endian and little endian

Assembly Language Programming

Assembly language is closely related to the processor under consideration. Thus, when writing assembler language programs the programmer needs to know four fundamental things:

- The problem to be solved: This is true for programming in general, not only for assembly
- The Instruction Set of the CPU: That is, what instructions are available to the programmer
- The CPU Register Set: What registers are available to the programmer to temporarily store and manipulate information
- The CPU Addressing Modes: What are the different ways of accessing data in memory

68K Instructions

This 68K processor family has many of the features present in commonly used computers

- Instruction Set: The 68000 has about 60 instructions, each with several variations. We are going to use only a reduced set that will be enough for our purposes

- Instruction Format: All instructions follow the format:

`operation.size source,destination`

where some of the operands may be implicit. For example:

`add.w d0,d1`

`move.b a0`

`bne next`

are all legal 68K instructions

68K Instructions

- Allowed operations include add, sub, mul, div, move ...
- The Source and the Destination are the locations of the operands, that may be an address in memory or a CPU register
- The data size is either:
 - b = byte, 8 bits
 - w = word, 16 bits
 - l = long word, 32 bits

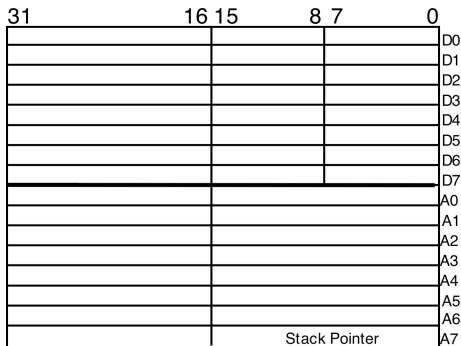
The Register Set

The 68000 has seventeen 32-bit registers and a 16-bit Status Register.

- Data Registers: There are eight general-purpose data registers, named d0 to d7
- Address Registers: There are eight address registers, named a0 to a7, specifically to store addresses
- Register a7 is the stack pointer (referred to as sp in assembly). This register may point to the *user stack* or the *system stack* (more on this later)

The Register Set

- Program Counter: stores the address of the next instruction
- Status Register: comprised of a user byte — the Condition Code Register (CCR) — and a system byte.



Program Counter

Status Register

The Stack

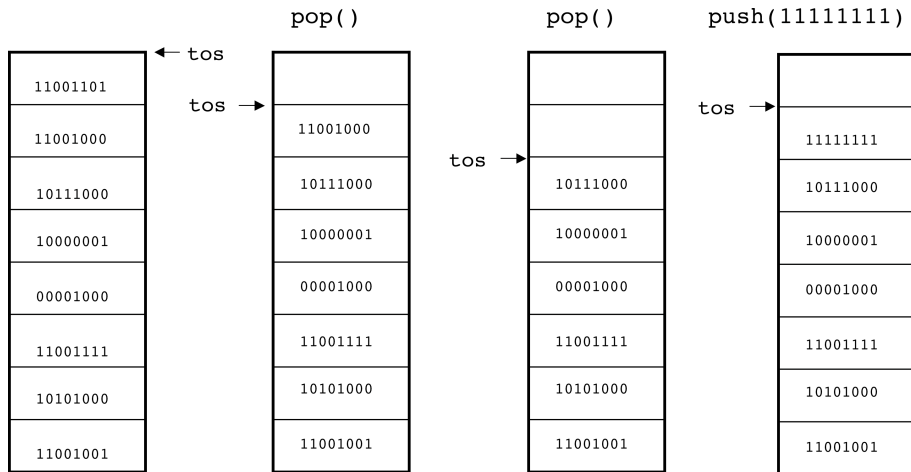


Figure: The stack structure

The Stack

- A stack is a set of memory locations used to store information temporarily
- Stack locations are sequential locations (side by side, like an array)
- The stack is managed at one end, the *top of the stack*, which is pointed to by the stack pointer.
- A stack is a LIFO (Last In First Out) structure, since the last element going onto the stack is the first element to come off the stack
- The only operations allowed at the top of the stack are:
 - `push()` an element on the stack
 - `pop()` an element from the stack
- A stack is similar to a stack of plates in a cafeteria, where the last plate to be put on top (pushed) is the first one to come off (popped)

Summary of the 68K Instruction Set

Table: Data Movement Operations

Instruction	Name	Data Size	Operation
exg	exchange	32	$R_x \leftrightarrow R_y$
move	move b, w, lw	8,16,32	$(EA)_s \rightarrow EAd$
swap	swap words	32	$W \leftrightarrow W$

Table: Compare Operations

Inst	Name	Data Size	Operation
cmp	compare	8, 16, 32	$D_n - S$, condition bits set
cmpa	compare address	16, 32	$A_n - S$, condition bits set
cmpi	compare immediate	8, 16, 32	$D - \#data$, condition bits set
cmpm	compare memory	8, 16, 32	$D - S$, condition bits set

Summary of the 68K Instruction Set

Table: Integer Arithmetic Operations

Instruction	Name	Data Size	Operation
add	add	8, 16, 32	$S + D \rightarrow D$
clr	clear	8, 16, 32	$0 \rightarrow EA$
cmp	compare	8, 16, 32	$D - S$
divs	divide signed	32 div 16	$D \text{ div } S \rightarrow RQ = D$
divu	divide unsigned	32 div 16	$D \text{ div } S \rightarrow RQ = D$
mul	multiply signed	$16 \times 16 \rightarrow 32$	$S \times D \rightarrow D$
mulu	multiply unsigned	$16 \times 16 \rightarrow 32$	$S \times D \rightarrow D$
neg	negate	8, 16, 32	2's complement
sub	subtract	8, 16, 32	$D - S \rightarrow D$

Summary of the 68K Instruction Set

Table: Branching Operations

Instruction	Name	Operation
bra	branch always	$PC + d \rightarrow PC$
bsr	branch to subroutine	$PC \rightarrow (SP), PC + d \rightarrow PC$
bcc	branch on condition	if TRUE then $PC + d \rightarrow PC$
beq	branch if equal	if E then $PC + d \rightarrow PC$
bne	branch if not equal	if NE then $PC + d \rightarrow PC$
bgt	branch if greater than	if GT then $PC + d \rightarrow PC$
bge	branch if greater than or equal	if GE then $PC + d \rightarrow PC$

Summary of the 68K Instruction Set

Table: Bit Manipulation Operations

Instruction	Name	Data Size	Operation
btst	bit test	8, 32	bit tested $\rightarrow Z$
bset	test bit and set	8, 32	1 \rightarrow bit
bclr	test a bit and clear	8, 32	0 \rightarrow EA
bchg	test a bit and change	8, 32	toggle bit

Table: Logical Operations

Instruction	Name	Data Size	Operation
and	logical and	8, 32	S and D \rightarrow D
or	logical or	8, 32	S or D \rightarrow D
eor	exclusive or	8, 32	S eor D \rightarrow D
not	logical not	8, 32	not(EA) \rightarrow EA

Sample Program

- The following is an assembly language program to add together the data bytes at memory locations \$3000, \$3001, and \$3002 and put the result into location \$3003.
- This is the equivalent of the code `result = data1 + data2 + data3`

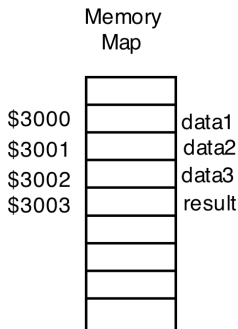


Figure: Memory map for sample program

Sample Program

```
org      $1000
move.l   #$7ffe,sp      ;initialise stack pointer
move.b   $3000,d0       ;get data from $3000
add.b    $3001,d0       ;add the next data
add.b    $3002,d0       ;add next data
move.b   d0,$3003       ;save result
move.b   #9,d0          ;return to simulator
trap     #15
end      ;end of code
```

Sample Program

- The `org` directive indicates to the assembler that the code is to reside in memory, starting at location `$1000`
- Next you will find the instruction mnemonics — `add`, `move` — each followed by the source and destination addresses. The data size for each instruction is byte, so the instructions end up in `.b`. Each part of the instruction line is delimited by spaces or tabs
- The final part of the instruction is a comment, which describe the action taken as part of the problem solution (not a description of the instruction itself). The general instruction format is:

```
mnemonic.size    source,destination    ;comment
```

Sample Program

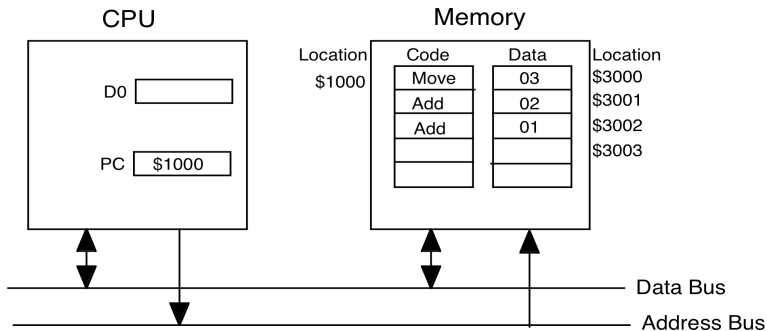


Figure: Running the sample program

NOTE: When creating this code you must put one or more spaces at the beginning of each line. These may be replaced by tabs

Sample Program

Let's trace the execution of the instruction:

```
move.b    $3000,d0        ;get data from $3000
```

As always, the instruction cycle is:

- ❶ **FETCH:** The address in the PC goes to memory via the address bus. The instruction in that location is transferred to the CPU instruction decoder via the data bus
- ❷ **DECODE:** The `move` instruction is decoded
- ❸ **EXECUTE:** The instruction and its parameters are identified:
 - ❶ The CPU sends address \$3000 to memory via the address bus
 - ❷ Memory transfers the data in memory location \$3000 to d0, via the data bus
 - ❸ The PC is incremented and the next cycle begins for the next instruction

Sample Program

- The program continues with the execution of the two add instructions. The contents of data register d0 will change from 03 to 05 then to a 06 as the program progresses

- The next instruction:

```
move.b    d0,$3003
```

puts the address, \$3003 onto the address bus and the data in d0 onto the data bus. The data will then be stored in location \$3003

- The two instructions:

```
move.b    #9,d0
```

```
trap      #15
```

exit the execution and return control to the user (more on this later)

The 68K Simulator

- To run a 68K programs on a PC the we will be using the EASy68K simulator
- In the labs you will first use the EASy68K editor/assembler to write and to assemble your programs.
 - Write your code in the editor
 - Go to the “Project” menu and click on the button for “Assemble”
 - The assembler will “pop up” a window with a button for the “Execute” option.
 - Use the EASy68K simulator to execute the assembled code
 - The simulator has it own set of controls for running and testing the code
- The use of the simulator will be covered during tutorials

Memory Alignment

- Often the processor moves around multibyte data, data made up of 2 or 4 bytes
- In the 68K both words and long words must start at even addresses
- An attempt to place a word or long word at an odd location will result in a “Bus Error” message
- Bytes can be at any address, but words and long words must be at even numbered addresses
- Words use two consecutive locations, whereas long words use four consecutive locations
- There may be some wasted space when words and long words are stored together, but the availability of cheap memory has made that waste less important

Memory Alignment

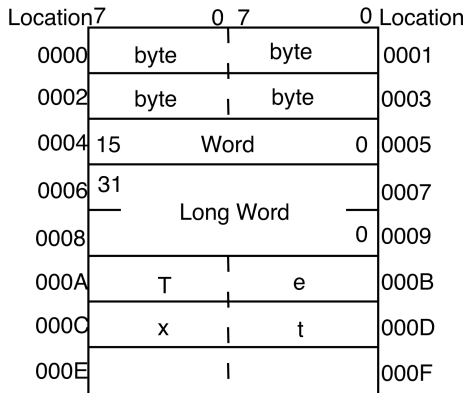


Figure: Multi byte data in memory

Endian

- Big endian: a two-byte integer is stored with the most significant byte first followed by the least significant byte like in the Motorola 68K
- Little endian: the least significant byte is at the lower memory location as in the Intel XX86 series microprocessors
- Big endian machines store the MSB at the lower address, while little endian machines store the LSB at the lower address
- There are advantages and disadvantages with both methods, but they are not too significant
- Big endian is a bit easier to read for humans, and this is very important for the interpretation of the 68K simulator dump

Endian

The 4-byte integer:

byte 3	byte 2	byte 1	byte 0
--------	--------	--------	--------

On a little endian machine it would be stored as:

base address + 0 = byte 0

base address + 1 = byte 1

base address + 2 = byte 2

base address + 3 = byte 3

On a big endian machine it would be stored instead as:

base address + 0 = byte 3

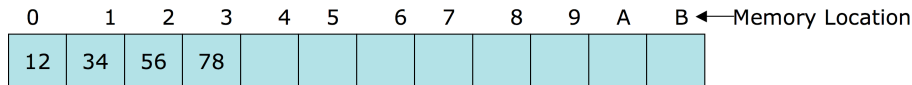
base address + 1 = byte 2

base address + 2 = byte 1

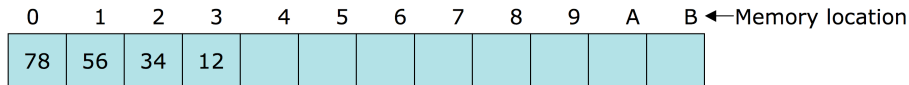
base address + 3 = byte 0

Endian

The figure illustrates the situation of the two approaches when storing a multi-byte quantity



Big Endian storing the 32 bit value \$12345678



Little Endian storing the 32 bit value \$12345678

Run time errors

- The most common run time error messages are bus errors, when trying to access an illegal address for example:
 - above \$7FFF, which is the top of memory
 - address errors when the CPU attempted to access a word or long word at an odd address
- To de-bug a program, study the register dump that occurs, and also look at the listing file, `filename.lst` to see which instruction caused the error
- Sometimes the error will be recognised by an odd numbered value in the address register
- The error is often a strange value in an address register
- Errors are corrected by modifying the source code, re-assembling and re-running the code

Sample Program

- Create, assemble, run and test the following program. Use the simulator and MM 2000 to put data in memory for the addition.
- This sample program adds together the word size data at locations \$2000, \$2002, \$2004 and save the result in memory location \$2006. Note that each number is 2 bytes, because the data is word size

```

org      $1000
move.l   #$7ffe,sp      ;init stack pointer
move.w   $2000,d3       ;get data 1
add.w    $2002,d3       ;add data 2
add.w    $2004,d3       ;add data 3
move.w   d3,$2006       ;save result
move.b   #9,d0          ;return to simulator
trap     #15            ;
end              ;no more to assemble

```

NOTE: Remember to put a space at the beginning of each line

Chapter 5

Data Representation

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this module, students should be able to:

- Explain binary multiplication operations, for unsigned and signed numbers
- Describe and correctly manipulate binary numbers with a fractional component
- Discuss the IEEE Standard 754 Floating Point Format, and properly find representations conforming with the format
- Describe alphanumeric codes such as ASCII and BCD, and convert between them and binary

Unsigned Binary Multiplication

- Binary *unsigned* numbers may be multiplied in a similar way to decimals. For example:

$$\begin{aligned}010\ 0011 * 101 &= (0010\ 0011 * 100) + (0010\ 0011 * 1) \\ &= 1000\ 1100 + 0010\ 0011 = 1010\ 1111\end{aligned}$$

- We can see in the example that:
 - multiplying by 10 (i.e. multiplying by 2) is shifting 1 bit to the left and adding 1 zero to the right
 - multiplying by 100 is shifting 2 bits to the left and adding 2 zeroes to the right
 - multiplying by 1000 is shifting 3 bits to the left, and adding 3 zeroes, etc
- So it is easy to multiply by shifting and properly adding after each shift

Unsigned Binary Multiplication

- The multiplication may go off the range of acceptable numbers:
 $1110\ 0011 * 101 = (1110\ 0011 * 100) + 1110\ 0011 = 100\ 0110\ 1111$
- The multiplication of two 8-bit numbers gives a 16-bit result, and the multiplication of two 16-bit numbers gives a 32-bit result
- We may use a similar algorithm to the one we use in decimal multiplication:

```

      1110 0011
    *         101
    -----
      11100011
      00000000
      1110001100
    -----
      10001101111
  
```

Sign Extension

- Often operations performed on 8 bits produce a 16 bit result
- Sometimes we need to extend a number from a lower to a higher number of bits — say 8 bits to 16 bits — without changing its value
- This is called *sign extension*. It is a way of extending the size in bits of a number, without changing its value
- To extend 0110 to 8 bits is quite simple: add four zeroes on the left 0000 0110 and the number does not change
- However, if we extend 1101 to 1111 1101 the value as a signed number does not change:

$$1101 = -2^3 + 2^2 + 2^0 = -8 + 4 + 1 = -3$$

and again

$$11111101 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 = -3$$

Binary Fractions

- We represent positive decimal fractions by using negative powers of the base 10:

Position Value:	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
decimal	.1	.01	.001	.0001	.00001

- Positive binary fractions may be represented by using negative powers of 2:

Position Value:	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
decimal	.5	.025	.125	.0625	.03125
fractions	$1/2$	$1/4$	$1/8$	$1/16$	$1/32$

Conversion of Positive Binary Fractions

- To convert a decimal fraction to a binary one, we repeatedly multiply by 2, copy and subtract the integer part until enough digits are obtained or a product is zero
- For example, convert $.637_{10}$

$.637 * 2 = 1.274$	[copy 1 down, and subtract 1]	1
$.274 * 2 = 0.548$	[copy 0 down, and subtract 0]	0
$.548 * 2 = 1.096$	[copy 1 down, and subtract 1]	1
$.096 * 2 = 0.192$	[copy 0 down, and subtract 0]	0
$.192 * 2 = 0.384$	[copy 0 down, and subtract 0]	0
$.384 * 2 = 0.768$	[copy 0 down, and subtract 0]	0
$.768 * 2 = 1.536$	[copy 1 down, and subtract 1]	1
$.536 * 2 = 1.072$	[copy 1 down, and subtract 1]	1
- Answer: $.637_{10} = 0.10100011_2$

Conversion of Positive Binary Fractions

- But multiplying by 256 a binary number results in shifting the fraction point 8 positions to the right, and dividing by 256 results in the fraction point shifted 8 positions to the left, so we may use an alternative method
- Considering 8 decimal places, using the same example again $.637_{10}$, we multiply and divide by 256:

$$(.637 * 256)/256 = 163.072/256$$

and this is approximately

$$163/256 = 1010\ 0011/256 = .1010\ 0011$$

which is the previous result

Conversion of Positive Binary Fractions

- The same method may be used to convert from binary fractions to decimal fractions
- Consider for example the binary number 0.0100 1001; this multiplied by 2^8 , results in 0100 1001, so:

$$\begin{aligned} 0.0100\ 1001 &= (0.0100\ 1001 * 256)/256 = 0100\ 1001/256 = \\ &= 73/256 = 0.28515625 \end{aligned}$$

Conversion of Positive Binary Fractions

- When a number has an integer part and a fractional part, both should be treated separately
- For example, to convert 59.637_{10} to binary, we first split the number as $59_{10} + .637_{10}$, and then:
 - convert 59_{10} to binary: 00111011_2
 - convert $.637_{10}$ to binary (previously done): $.10100011_2$

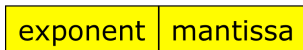
Therefore: $59.637_{10} = 00111011.10100011_2$

- Addition operations are also performed separately, taking care of the carry out of the decimal part into the integer part, as in:

$$00111010.10100011_2 + 01000100.10001100_2 = 01111111.00101111_2$$

Floating Point Representation

- Binary integers are of fixed and limited size, and therefore it is not possible to represent very small negative or very large positive values.
- Floating point numbers allow a much wider range of numbers to be represented
- However, floating point numbers are only approximations of the actual value, and lose accuracy with the approximate representation
- Format:
 $a * r^{exp}$, where a = mantissa, r = radix/base, e = exponent
- In the computer two fields are stored as depicted in the figure
- The exponent and mantissa can use any of the binary systems
- The radix is not stored as it is a known value



Floating Point Representation

- Floats can represent a wide range of numbers, but there is almost always some error because of the distance between each number in the representation scheme
- The conversion between the actual number and the computer representation is not always exact, even when storing an integer
- When an integer number is stored as a float, it may suffer from loss of accuracy due to the representation
- The integer number 250 when stored as an integer is 1111 1010, an exact representation of 250
- However, if we store 250 as a float with the previous scheme, we get only an approximation

IEEE Standard 754 Floating Point Format

- In 1985 the IEEE introduced the IEEE Standard 754 Floating Point Format to represent floating point numbers
- This IEEE standard is widely used in modern processors
- There are three different precisions in use:
 - Single precision: 32 bits in length, as follows:
 - 1 bit sign
 - 8 bit exponent in excess 127 format
 - 23 bit mantissa
 - Double precision: 64 bits in length, as follows:
 - 1 bit sign
 - 11 bit exponent in excess 1023 format
 - 52 bit mantissa
 - Extended precision: 80 bits in length, only used internally by the computer arithmetic unit to reduce rounding errors.

IEEE Standard 754 Floating Point Format

To store an integer as an IEEE 754 Standard, say store 10.5_{10} to Single Precision Format

- 1 Convert to binary
 - 1 $10 = 1010_2$
 - 2 $.5 = .10_2$
 - 3 $+10.5 = 1010.10_2 \times 2^0$
- 2 Normalise (write with a significant bit 1 before the dot) 1.01010×2^3 The 1. is not stored in the computer; this increases room for precision by one bit. The 1 is put back in arithmetic operations.
- 3 Exponent is +3 in excess 127 format $= 127_{10} + 3_{10} = 130_{10} = 1000\ 0010_2$
- 4 Mantissa $= 0101\ 0000\ 0000\ 0000\ 0000$
- 5 Sign $= 0$, positive
- 6 IEEE $= \%0100\ 0001\ 0010\ 1000\ 0000\ 0000\ 0000\ 0000 = \$4128\ 0000$

IEEE Standard 754 Floating Point Format

Some remarks:

- In single-precision, when the exponent is 255
 - if the significand is zero, the quantity is \pm -infinity,
 - the significand is non-zero the quantity is Not a Number (NaN), (usually means error)
- In double-precision, when the exponent is 2047
 - if the significand is zero, the quantity is \pm -infinity,
 - the significand is non-zero the quantity is Not a Number (NaN), (usually means error)

Example

- Let's assume that we need to convert \$40C4 0000 to decimal.
 - \$40C4 0000 = %0 100 0000 1100 0100 0000 0000 0000
 - Sign = 0 (positive)
 - Exponent = 1000 0001 = $129_{10} - 127_{10} = 2$
 - Significant = %1.100010000 (Note that the 1 is put back)
 - Result = $1.10001 * 2^2 = 110.001_2 = 6.125_{10}$
- The loss of accuracy often results in errors in arithmetic calculations. Assume that the computer handles 4 bit decimal numbers, and the decimal calculation by hand

$$543.57 + 1.862 = 545.432$$

- If we align both numbers we have $0.5435 * 10^3 + 0.1862 * 10^1$
- To add the two numbers we must make the exponents equal:
 $0.5435 * 10^3 \rightarrow 0.5435 * 10^3$
 $0.1862 * 10^1 \rightarrow 0.0018 * 10^3$
- Adding them together, we have: $0.5453 * 10^3 = 545.3$. Since the calculation by hand is 545.432, the difference is significant

ASCII Code

Aside from numbers, computers manipulate alphanumerical characters. The most common codes in use are ASCII, EBCDIC and Unicode codes. The most often used is the ASCII (American Standard Code for Information Interchange) code

- 7 bit code used world wide, usually stored in a computer in an 8-bit byte with the most significant bit left as 0¹
- there are 128 different characters
- 10 digits, 0-9
- 26 uppercase letters
- 26 lowercase letters
- 7 punctuation marks
- 34 control characters
- 26 other characters

¹Sometimes an 8-bit version is used, by either keeping the number of 1s even (even parity) or odd (odd parity)

ASCII Table

	0	1	2	3	4	5	6	7
0	NUL	DLE	sp	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	,	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

ASCII Collating Sequence

The order assigned within a code is called the *collating sequence*. For ASCII, we have:

A 100 0001
B 100 0010
C 100 0011
D 100 0100

0 011 0000
1 011 0001
2 011 0010

Thus characters and text strings can be compared using their ASCII values:

A < D
FRED < FREE
0 < 7
X < e
A < a

Other Codes

- Extended ASCII (not standard): the original ASCII was extended to 256 symbols to represent characters not present in the ASCII standard. Unfortunately, this extension is not a standard
- EBCDIC: an 8-bit character code invented by IBM, comprising 256 characters
- Unicode: this is a newer code to cater for many of the world's languages, such as Asian languages that have many symbols:
 - It uses 16 bits to represent each character
 - A subset corresponds to ASCII, that is Unicode extends the ASCII code
 - The Java programming language uses Unicode to represent characters. The complete set may be found at <http://www.unicode.org/charts/>

Binary Coded Decimal

- BCD Binary Coded Decimal numbers are used in systems where only numbers are being entered
- Although BCD arithmetic can be performed without the need to convert to binary, many systems perform the conversion to implement the operation so they can use the same circuits
- The BCD table is as follows:

0= 0000	5= 0101
1= 0001	6= 0110
2= 0010	7= 0111
3= 0011	8= 1000
4= 0100	9= 1001

Codes 1010 to 1111 are not used.

- **Example:** 1,295= %0001 0010 1001 0101
To make it more efficient the code may be packed, two symbols per byte

ASCII to Binary

- This conversion is performed by a computer to convert ASCII keyboard keys to binary, when arithmetic is to be carried out. The keyboard reads ASCII '1' = 0011 0001, but if the program needs the integer 1, what it needs is %0000 0001. To convert '1' to binary 1 for arithmetic, subtract %0011 0000 which leaves %0000 0001
- Similarly, when printing a binary number such as %0000 1001, (= 9), the computer must add %0011 0000 to give %0011 1001 which is ASCII for '9'. Converting multi-digit binary numbers, such as %1110 1110 to the corresponding sequence of ASCII characters '238' is more complex

String to Integer

- Converting '238' to its integer equivalent is not too difficult
- It is used very often, since strings such as '238' are entered via the keyboard to input integer numbers. This is what the Java method `Integer.parseInt()` does
- Via the keyboard, the characters come one by one in a sequence 2, 3, 8. To construct the binary number, we pick the characters one by one, multiply by 10 and add them each time as follows:
 - first character $\rightarrow 2$
 - multiply by 10 $\rightarrow 2 * 10 = 20$
 - add second character $\rightarrow 20 + 3 = 23$
 - multiply by 10 $\rightarrow 23 * 10 = 230$
 - add third character $\rightarrow 230 + 8 = 238$

String to Binary

- In the case of binary, we may use the same algorithm, but the operations are binary
- It is not immediate to multiply by 10 in binary, so we use a little maths:
 $'2' - '0' = 0011\ 0011 - 0011\ 0000 = 0000\ 0011 = 2_{10}$

and then use the algorithm, remembering that multiplying by 10 is the same as multiplying by 8 and then by 2 and adding:

- 1 $2_{10} * 10_{10} = 2_{10} * 8_{10} + 2_{10} * 2_{10} = 00000010 * 1000 + 00000010 * 10 = 00010000 + 00000100 = 00010100 (= 20_{10})$
- 2 $20_{10} + 3_{10} = 23_{10} = 00010100 + 00000011 = \dots$
- 3 $23_{10} * 10_{10} = \dots$

Exercise: Finish this off

Student Exercises

- ❶ Convert the text string 'Yoko's in Tokyo' to 7-bit ASCII string
- ❷ Convert the string 'Mary's eyes are blue' to an 8-bit ASCII string
- ❸ Convert the 7-bit ASCII code to text:

```
10001111101111110111111001000100000  
1101101110111111100101101110110100111011101100111010  
0000100110111100100100000101001111011011101001111010
```

- ❹ Convert the ASCII 8-bit code to text:

```
01001000011011110110110001101001  
01100100011000010111100100101110
```

- ❺ Convert the string '137' to its equivalent binary

Chapter 6

Computer Organisation-Assembly

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Explain memory alignment and its consequences
- Explain what exceptions are, and discuss software and I/O interrupts, including the interrupt mask
- Explain what Direct Memory Access and how it helps to make I/O more efficient
- Describe the Status Register and its relationship with branching
- Create programs with branching instructions

Sample Program

```

org            $1000
move.l        #$7ffe,sp        ;init stack pointer
move.l        data1,d2         ;get data1
add.l         data2,d2         ;add data2
add.l         data3,d2         ;add data3
move.l        d2,result        ;save result
move.b        #9,d0            ;stop
trap          #15

```

```

org            $2500
data1          dc.l            1        ;initialised = 1
data2          dc.l            2        ;initialised = 2
data3          dc.l            5        ;initialised = 5
result         ds.l            1        ;long word for
end            ;the result

```

Sample Program: Labels

- This program adds together three long words at memory locations \$2500, \$2504, \$2508 and saves the result in location \$250C
- This program uses labels, which are symbolic references to addresses
- We can use the place where a label is defined — the *location counter* — to refer to the address
- The code now allocates four memory locations for the data. The locations `data1`, `data2` and `data3` are long words reserved and initialised by the system, whereas `result` is allocated but not initialised
- The labels `data1`, `data2`, `data3` and `result` represent the locations of the data, the computer identifies the locations by the labels
- We don't know the addresses of the data — and we should not care — but the computer does
- We should always use labels to avoid 'hard wiring' addresses in our code

Sample Program: Labels

- The code starts as before by initialising the stack pointer, and moving the first data item into register d2
- The sum is accumulated in d2
- Then both data2 and data3 are added to d2, and finally the sum is placed into result
- After that, the program exits

NOTE: spaces before instructions, no spaces before labels

Memory Alignment

- Certain memory organisations require objects to be located at particular addresses in memory
- The processor expects words or long words to start at particular addresses in memory
- The 68K requires that all words and long words are stored at even addresses in memory (i.e. LSB is 0): a *word boundary*
- If not, the 68K processor throws an exception: *hard alignment*
- That is why we use the line:

```
move.l    #$7ffe,sp
```

to initialise the stack pointer on an even address in memory, instead of the very top of available memory `#$7fff` which is an odd address

- Other processors (e.g. Intel 8086) accept a misalignment: *soft alignment*. There is a performance penalty

Exceptions

Exceptional behaviour (exceptions) may occur:

- External exceptions: interrupts, interruptions by external devices
- Program exceptions:
 - ① an operation exception, such as 'attempt to divide by zero' or a floating-point operation that gives overflow; the operation must be stopped
 - ② *illegal instructions* and *unimplemented instructions*: are detected by the decoding system
- Instruction/Data access exceptions: raised by the memory management system:
 - ① an attempt is made to access an instruction or data that is not physically present; it is a page fault, issued by the virtual memory system
 - ② or it is outside of the region allocated to a process. The exception is a response to an attempt to access memory beyond the process allocated area: common cause of 'Bus Error'

Exceptions

Software interrupts:

- Requests to the operating system to perform a task, e.g. writing characters to a file on disk
- These operations are too delicate and dangerous to leave them to the programmer, so they are not implemented as a normal program instruction
- An operating system provides ready-to-run software instructions — *privileged instructions* — and routines to perform these operations
- These system calls, traditionally called *traps*, and execute in *supervisor mode* to be able to access the hardware

Exception Handling

When an interrupt is detected the processor stops execution, and attends to the interrupt²:

- ① An interrupt number or type is obtained (from the device, the software, etc.)
- ② This number is used to access a table in the lower section of memory
- ③ The table entry points to the address of the *interrupt routine* (*exception handler*)
- ④ The system produces a *context switch*:
 - ① it saves the state of the executing process on the system stack
 - ② it dispatches the interrupt routine
 - ③ the routine executes (usually) until the end
 - ④ restores the original state and then returns to normal program execution

²This description corresponds to *vectored interrupts*.

Exception Handling

After executing the exception handler, the program either:

- Exits if the processor cannot continue executing (a *fatal exception* such as divide by zero) . The exception handler will display an error message and exit
- Continues executing: non-fatal exception. The exception handler may fix up or skip over the error and return control to the running code. It is responsibility of the programmer to deal with non-fatal exceptions properly

The task of saving the processor state is *critical*, the task itself cannot be interrupted. Processors have special instructions to disable interrupts during a critical task

Traps or Interrupts

Discussed in greater detail in the *Operating Systems* course.

- Which trap number does what is dependent on the operating system in use.
 - MS DOS used trap 21H as the primary interrupt
 - The EASy68K simulator uses trap 15 (0FH)
 - Sim68K used trap 14 (0EH)
- The trap number is an *index* into an array of addresses of software routines
- The location of the array is “hard-coded” in the CPU

Interrupt Vector Table

The proper name for the address array is the *interrupt vector table*

- Address of the handler for each interrupt
- Typically has 256 entries (one byte used for the interrupt number)
- Entry size is dependent on the address size in the system
- Data is passed to the handler in the registers

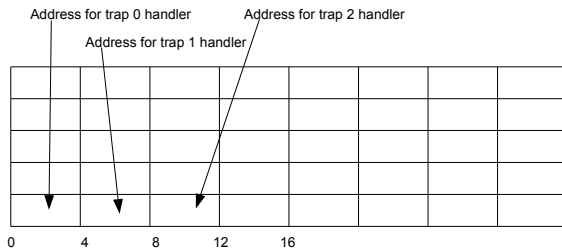


Figure: Interrupt Vector Table

Traps

- In the 68K, programmer interrupts (traps) are implemented by initialising (move) D0 with a function number and then executing the trap instruction
- The operating system uses the function number provided to determine which routine to execute
- Some useful functions are:
 - #5 ; INCH, reads a keyboard ascii key into D1.B
 - #6 ; OUTCH, outputs ASCII in D1.B to the screen
 - #9 ; EXIT, exits back to the simulator
- Please note that a0 and a1 may be changed by a trap function. This is implementation-dependent, so it is not possible to anticipate

Traps

The following program inputs and displays (echoes) a keyboard character to the display. Create and test.

```
org      $1000
move.l   #$7ffe,sp      ;init stack
; read the keyboard
move.b   #5,d0          ;prepare to read a character
trap     #15            ;get the character
; display the key
move.b   #6,d0          ;prepare to display
trap     #15            ;display the character
; stop
move.b   #9,d0          ;prepare to exit
trap     #15            ;exit
end
```

Fetch-Decode-Execute

Recall that this is the cycle that the computer follows when running a program:

- ➊ Fetch:
 - ➊ Copy the PC onto the MAR
 - ➋ Fetch the instruction in the MAR and place it in the IR
 - ➌ Increment the PC to point to the next instruction
- ➋ Decode: Interpret the instruction, and if there is a need to fetch data place the address in the MAR
- ➌ Execute:
 - ➊ Use the address in the MAR to get the data (if required) and place it in the MBR
 - ➋ Execute the instruction

This changes by checking for interrupts at the beginning of each cycle

Interrupts

If an interrupt is issued by an I/O device, the CPU:

- 1 stops executing after the current instruction
- 2 attends to the interrupt
- 3 identifies the interrupting device
- 4 invokes the interrupt handler, and the sequence proceeds as described before

Interrupt Priority

In I/O the *interrupt priority* is very important:

- Each interrupt has a priority
- Interrupts of lower priority are allowed to be interrupted by those of higher priority
- Less urgent interrupts are interrupted by more urgent ones
- In I/O, priority depends on the device: devices with shorter *response deadlines*, such as a network connection, are serviced first, to minimise the risk of losing data
- Interrupts associated to a device such as a keyboard are of lower priority than network interrupts, for example

Interrupt Mask

- An interrupt routine may itself be interrupted mid-execution
- Another interrupt routine will be dispatched: *nested interrupts*
- Bits 8, 9 and 10 of the Status Register are the *interrupt mask*:
 - The bits that indicate the priority of the current interrupt
 - If the mask value is 100 (level 4), it may not be interrupted by interrupts with priorities 000, 001, 010, 011, 100
 - Only interrupts with priority greater than 4 — i.e. 101, 110 and 111 — would be serviced
 - When an interrupt is serviced, the interrupts mask is changed to the new level
 - If an interrupt level 5 is being serviced, the mask in the status register is changed to 101

Non-maskable Interrupts

Most processors support *non-maskable interrupts*:

- Interrupts that cannot be ignored and have to be serviced immediately
- This is to attend to very urgent tasks, such as the computer going down or as a response to some urgent outside world event
- The 68K uses level 7 as a non-maskable interrupt, an interrupt level 7 is always serviced by the processor
- In the unlikely event in which the processor detects an interrupt level 7 while servicing another interrupt level 7, the new interrupt is serviced

Direct Memory Access: DMA

- Even the fastest CPU will struggle to cope with the speed of I/O
- The CPU wastes cycles moving data around
- Solution: allow simple programmable device controllers to read from and write to memory directly: *Direct Memory Access (DMA)* devices
- The CPU provides the device controller with the location and the number of bytes to be transferred, and the destination device if it is a write or the memory address if it is a read
- The CPU then signals to the controller that the information for the transfer is ready, and k.pdf going
- From then on the CPU and the DMA device compete for the same bus, but only one of them at the time may be the *bus master*.
- The DMA becomes the bus master for the period of a read or write, performs the required transfer, and then frees the bus
- The DMA device interferes steals cycles from the processor, but this does not significantly affect performance since there is no context switch

I/O Devices

- In small computer systems it is common to provide each device with an interrupt request line (and corresponding number)
- When a device needs to interrupt the CPU it raises its I/O interrupt line
- When the CPU is ready to process the interrupt, it raises the Interrupt Acknowledge (INTA) line to start the process
- Typically there are more devices than available interrupts, so interrupt numbers must be shared
- Devices unlikely to clash such as a printer and a scanner may share the same number, but there may be conflicts between others
- The I/O controller makes sure that all connected devices work without conflict

Ports

- Ports are the contact points of the computer communication with the outside world
- Allow the transfer of data between the computer and external devices such as a mouse, a hard disk or a printer
- There are two types of ports, parallel and serial:
 - In parallel communication several bits (e.g. eight) are transmitted in parallel at the same time: faster than serial, more error prone over long distances
 - In serial communication bits are transferred one at a time in a sequence, it may be used over any distance

Serial Communication Standards

Lately, some standards for fast serial communication have emerged:

- *USB, Universal Serial Bus:*
 - supports 'Plug and Play' and 'Hot Plugging' ('Hot Swapping')
 - USB provides interconnection between PC's and a wide variety of peripherals
 - Version 1.1 of this standard supports data rates of up to 12 megabits per second while version 2.0 supports rates up to 480 megabits per second.
 - USB supports bi-directional data transmission, and also includes +5V and ground wires for supplying power to devices such as iPods and portable hard drives
- *FireWire:*
 - Very fast peripheral standard, to connect the computer with multimedia peripherals
 - Up to between 400 megabits per second and 800 megabits per second in normal operation
 - Is an implementation of the high-speed serial data bus defined by the IEEE 1394-1995, IEEE 1394a-2000, and IEEE 1394b standards
 - Simplified cabling, hot swapping, and transfer speeds of up to 800 megabits per second (on machines that support 1394b).

Program Flow

Sequence: The statements are executed in sequence, one after the other. This is the normal operation of the processor

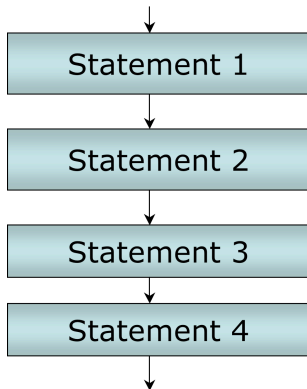


Figure: Sequential statements

Program Flow

A choice statement: the equivalent of an `if ...else` statement. A different execution path is taken according to the true/false value of a condition

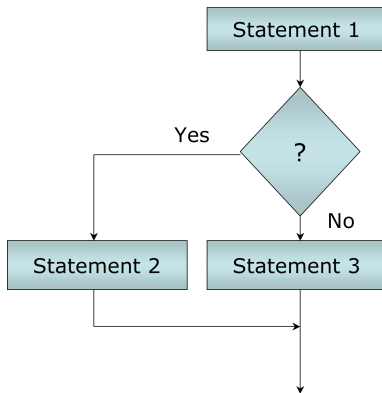


Figure: A choice: an `if...else`

Program Flow

Iterations where the loop is executed at least once. This is equivalent to a do...while loop

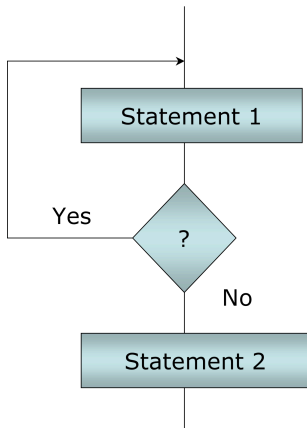


Figure: A do...while loop

Program Flow

Iterations where the loop may not be executed at all. This is equivalent to a while loop

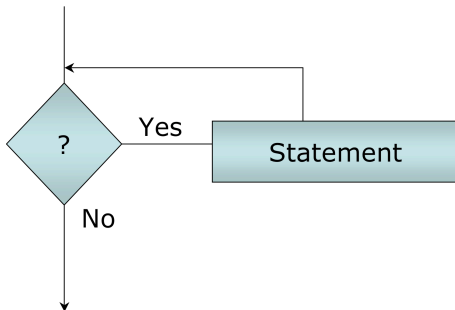


Figure: A while loop

Status Register

- Status Register, a 16-bit register where the lower byte represents the result of operations within the CPU for programmer's use, and the higher byte is used by the system to control the running of programs
- The Condition Code Register CCR may be inspected to find out whether the previous instruction resulted in zero, negative, overflow, etc
- The higher byte controls some execution parameters such as the interrupt mask, supervisor mode, tracing mode
- Condition Code Register: users' byte
 - C = carry
 - V = overflow
 - Z = zero
 - N = negative
 - X = extend

Status Register

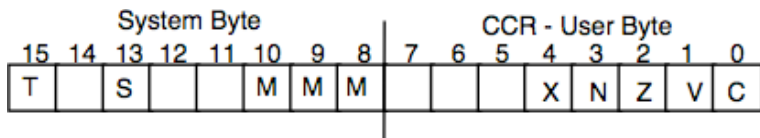


Figure: Status Register

Branching

- The CCR is related to branching in programs
- Almost every operation results in a change of the CCR bits
- Programmers may test whether the previous operation set the zero bit, (set $Z = 1$) and if so, resume program execution at a different location
- If a variable counter is initially 3 and it is decremented 3 times, at the end of the process counter will be zero and then $Z = 1$
- Thus, it is possible to execute a set of statements a number of times and exit at the end of the loop

Branch Instructions

The most common are:

- **beq:** Branch If Equal. Take the branch if the previous instruction set the zero bit
- **bne:** Branch If Not Equal. Branch if the result of the previous instruction reset the zero bit
- **bge:** Branch If Greater or Equal. Branch if the result of the previous instruction was greater or equal to zero
- **bgt:** Branch If Greater. Branch if the result of the previous instruction was greater than zero
- **bra:** Branch Always: Always takes the branch
- **bcc:** Branch If Carry Bit is Clear ($C=0$)
- **bcs:** Branch If Carry Bit is Set ($C=1$)

An if statement

Assume that we have an integer number in the lower byte of d1. The following code implements an if in assembly:

```

        cmp.b    #5,d1        ; compare d1 with 5
        bge      skip        ; if d1 >= 5 skip if
        .....              ; if statements here
        .....
skip    .....              ; condition was false

```

The code above implements:

```

if (d1 < 5)
    // if segment;

```

An if...else statement

Assume that we have an integer number in the lower byte of d1. The following code implements an if...else in assembly:

```

        cmp.b    #5,d1      ; compare d1 with 5
        blt      if         ; if d1 less than 5 branch to if
        .....          ; else here
        .....
        bra      skip       ; skip the if clause
if      .....          ; if segment
        .....
        .....
skip    .....
        .....
```

The code above implements:

```

if (d1 < 5)
    // if segment;
else
    // else segment;
```

A do...while statement

The following assembly code implements a do...while loop.

```

        move    #0, d1
next    add     #1, d1
        .....
        .....
        cmp     #7, d1
        blt     next
        .....

```

This is equivalent to the do...while loop:

```

d1 = 0;
do
{
    .....
    .....
} while (d1 < 7);

```

A while statement

The following assembly code implements a while loop.

```

        move    #0, d1
next    cmp     #7, d1
        bge     follow
        .....
        .....
        add     #1, d1
        bra     next
follow  .....

```

This is equivalent to the while loop:

```

d1 = 0;
while (d1 < 7)
{
    .....;
    .....;
}

```

Loops

- In high level languages it is common to write loops with an increasing counter — e.g. `for (i = 0; i < 5; i++)`
- Since branch instructions are often related to zero, loops in assembly are almost always implemented with a decreasing counter, say from 4 down to 0
- In the 68K these instructions are usually used together with an extra addressing mode that makes things easier for the programmer

is_digit

The following code may be used to detect whether a character in a register — d0 — is a digit, and process accordingly. It assumes the existence of two code segments labeled `no_digit` and `yes_digit`:

```
is_digit      cmp.b    '0',d0      ; every ASCII less
              blt      no_digit    ; than '0' is not a digit
              cmp.b    #'9',d0
              ble      yes_digit   ; between '0' and '9' is
              bra      no_digit    ; everything > '9' isn't
```

is_word_char

The following code determines whether a character in `d0` is a word character (either a digit or a letter character) it assumes two code segments labeled `iwc_yes` and `iwc_no`:

```
is_word_char    cmp.b    #'0',d0        ; ASCII less than
                blt      iwc_no          ; '0' is not word char
                cmp.b    #'9',d0
                ble      iwc_yes         ; >= '0' and <= '9' is
                cmp.b    #'A',d0
                blt      iwc_no          ; > '9' and < 'A' isn't
                cmp.b    #'Z',d0
                ble      iwc_yes         ; >= 'A' and <= 'Z' is
                cmp.b    #'a',d0
                blt      iwc_no          ; > 'Z' and < 'a' isn't
                cmp.b    #'z',d0
                blt      iwc_yes         ; >= 'a' and <= 'z' is
                bra      iwc_no          ; (> 'z' isn't)
```

if ...else

The previous examples implement an if ...else logic. In general, the implementation of this logic is standard. For example, if we want to implement the logic:

```
if (d1 > 0)
    a = 4;
else
    a = 5;
```

we may write:

```

                                cmp.b    #'0',d1    ;
                                bgt      four        ; jump to if clause
                                move.b   #5,a        ; else clause
                                bra       ahead       ; skip over
four                            move.b   #4,a        ;
ahead                          .....

```

Exercise: provide an alternative implementation of an if ...else

Chapter 7

Assembly Language-Digital Logic

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe more assembly addressing modes and their relationship with loops
- Present the principles of Boolean Algebra, and its operators
- Simplify Boolean expressions using Boolean Algebra rules
- Describe the basic logic gates

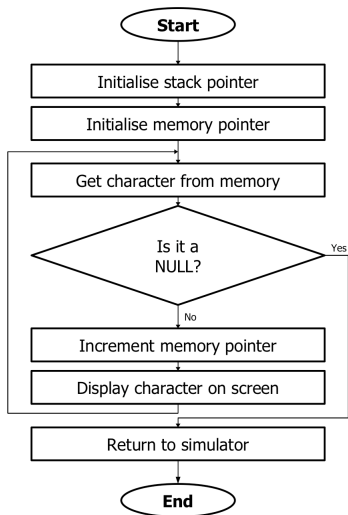
Address Register Indirect

- To better make use of loops another 68K addressing mode is convenient:
(an) Address Register Indirect
- With (an) the CPU uses the specified register to supply the address of the data in memory, as in:

```
move.b (a3),d0
```
- Rather than the contents of a3 being the data to be accessed, the content of a3 is the address of the data to be accessed
- By putting this instruction in a loop and then incrementing the content of a3 we increment the address of the data
- On each iteration of the loop, the CPU will access incremental locations each time around the loop
- A register such as a3 may be used to sequentially access all the characters in a string

Flow Chart

The flow of control structures make possible to create a flow chart to illustrate the logic of a program:



Prints String in Memory With a Loop

3

```

        org      $1000          ;code space
        move.l   #$7ffe,sp      ;always first instruction
        move.l   #string,a3     ;initialises a3 = $2000
next     move.b   (a3),d1        ;get character
        beq      exit          ;terminate if 'null'
        add.l    #1,a3          ;point to next char
        move.b   #6,d0          ;display char
        trap     #15            ;display it
        bra      next          ;loop until null
exit     move.b   #9,d0          ;prepare to return
        trap     #15            ;exit back

data     org      $2000          ;data space
string   dc.b     'this is the beginning,',cr,lf
        dc.b     'and this is the rest',cr,lf,null
lf       equ      $0a           ; line feed
cr       equ      $0d           ; carriage return
null     equ      00            ; null
        end

```

³Note: EASy68K has built-in string output functions but we are not using them yet

Loops in Assembly

The example implements a loop to execute a set of instructions until a certain condition (the end of the string) is true:

```
next    move.b    (a3),d1        ;get character from string
        beq       exit          ;terminate when 'null'
        add.l     #1,a3         ;increment a3 to next char
        move.b    #6,d0         ;display the char in d1
        trap      #15           ;display it
        bra       next          ;loop until null character
```

This is the assembly version of a while loop:

```
while (not the end of the string)
{
    get character;
    display character;
}
```

Loops in Assembly

Other high-level loops may also be translated to assembly with branching:

```
for (i = 0; i < 5 ; i++)
    statements;
```

This loop may be implemented in assembly as follows:

```

next      move.b      #4,d2          ; set up d2 as counter
          beq         exit          ; terminate when is zero
          .....          ; statements here ...
          .....          ; statements here ...
          sub         #1,d2         ; decrement counter
          bra         next          ; loop next
exit
```

Loops in Assembly

- Be careful though, because the value of d1 is the complement of the value of i in the loop, when i = 0, d1 = 4, when i = 1, d1 = 3, etc
- Alternatively, this type of loops may be implemented as follows:

```

next      move.b      #5,d2          ; set up control counter
          sub         #1,d2          ; decrement counter
          .....        ; statements here ...
          .....        ; statements here ...
          beq         exit           ; terminate when is zero
          bra         next          ; loop next
exit
```


Loops in Assembly

- Again, be careful because in the for loop i goes from 0 to 4, whereas d1 goes from 4 to 0
- Here is another version:

```

next      move.b      #4,d2          ; set up control counter
          .....
          .....                    ; statements here ...
          .....                    ; statements here ...
          beq         exit           ; terminate when is zero
          sub         #1,d2          ; decrement counter
          bra         next           ; loop next
exit
```

Loops in Assembly

And another version:

```

next      move.b      #5,d2          ; set up control counter
          beq         exit          ; terminate when is zero
          sub         #1,d2         ; decrement counter
          .....         ; statements here ...
          .....         ; statements here ...
          bra         next          ; loop next
exit
```

Although all these versions are pretty similar, they are not equivalent, and one may be better suited than others under certain circumstances

Boolean Operators

- Boolean algebra is appropriate for computing, since it is based on the notion of true or false
- Operands that are either true or false, and operations that combine those operands
- Usually, true is represented as 1, and false as 0
- OR operator: returns true when either operand is true

Table: OR operator

OR	+
$0 + 0$	$= 0$
$0 + 1$	$= 1$
$1 + 0$	$= 1$
$1 + 1$	$= 1$

Boolean Operators

- AND operator: returns true when both operands are true

Table: AND operator

AND	.
0 . 0	= 0
0 . 1	= 0
1 . 0	= 0
1 . 1	= 1

- NOT operator: returns the opposite value

Table: NOT operator

NOT	–
0	= 1
1	= 0

Boolean Operators

- Operators are given a precedence, like in normal arithmetic $2 + 3 * 4$ must be interpreted as $2 + (3 * 4) = 14$, rather than $(2 + 3) * 4 = 20$, since the multiplication 'binds stronger' than the sum
- When one operator has a higher precedence than another, it also means that it binds stronger
- As it may be seen in the table, the highest precedence is the $()$, followed by NOT, AND and OR.

Table: Operator precedence

$()$	highest
NOT	
AND	
OR	lowest

NOTE: often the notations \bar{a} or $/a$ are used instead of NOT(a)

Boolean Operators

- These Boolean operations have particular characteristics that are different to the ones we are used to, such as:

Table: Boolean operators properties

$0 + X = X$	$0 \cdot X = 0$	$X = \overline{\overline{X}}$
$1 + X = 1$	$1 \cdot X = X$	
$X + X = X$	$X \cdot X = X$	
$X + \overline{X} = 1$	$X \cdot \overline{X} = 0$	

- With binary data, we can prove an identity by going through all the possible situations, verifying that the equality holds true at all times
- This is called a *truth table* and it is used very often

Truth Table

As an example, let us prove the identity $X + \bar{X} = 1$ using a truth table:

Table: A truth table for identity $X + \bar{X} = 1$

X	\bar{X}	$X + \bar{X}$
0	1	1
1	0	1

Rules

Other identities involving more than one variable are also useful:

Table: Multi-Variable Rules

Commutative Law	$A + B = B + A$
Associative Law	$A + (B + C) = A + B + C = (A + B) + C$ $A (BC) = ABC = (AB) C$
Distributive Law	$A (B + C) = AB + AC$
Others	$A + AB = A$ $A (A + B) = A$ $A + BC = (A + B)(A + C)$ $A + \overline{A}B = A + B$ $AB + \overline{A}B = B$

de Morgan's Theorems

In addition, the following two identities are very useful:

- $\overline{A + B} = \overline{A} \cdot \overline{B}$

- $\overline{A \cdot B} = \overline{A} + \overline{B}$

NOTE: It is very important to note that:

- $\overline{A + B} \neq \overline{A} + \overline{B}$

and that

- $\overline{A \cdot B} \neq \overline{A} \cdot \overline{B}$

de Morgan's Theorems

In fact, intuitively, the de Morgan's Theorems actually say:

"The NOT of an OR is the AND of the NOTs",
and

"The NOT of an AND is the OR of the NOTs"

This is simple logic. Consider for example the second Theorem and reason as follows:

"If and $A \text{ AND } B$ is false, then either A or B is false, and therefore $\bar{A} \text{ OR } \bar{B}$ is true"

Conversely,

"If either A or B is false, then $A \text{ AND } B$ cannot be true, and therefore $A \text{ AND } B$ is false"

Simplification

Given a Boolean expression, it may be very important to simplify it as much as possible, since many of these expressions are to be implemented in digital logic.

Example:

$$\begin{aligned}
 Z &= A.B.C + A.\bar{B}.(\overline{\overline{A.C}}) = \\
 &= A.B.C + A.\bar{B}.(\overline{\bar{A} + \bar{C}}) = \\
 &= A.B.C + A.\bar{B}.(A + C) = \\
 &= A.B.C + A.\bar{B}.A + A.\bar{B}.C \\
 &= A.B.C + A.\bar{B} + A.\bar{B}.C \\
 &= A.C(B + \bar{B}) + A.\bar{B} \\
 &= A.C + A.\bar{B}
 \end{aligned}$$

; de Morgan's theorem

$$; A = \bar{\bar{A}}$$

; multiply out

; since $A.A = A$

; common factor AC, $B + \bar{B} = 1$

Simplification Example

Trace the operations of the following example, and make sure that you can relate each operation to the rules above:

$$\begin{aligned}
 Z &= \overline{(\overline{A}.\overline{B} + C)} + (B.C) \quad (= \overline{\overline{A}.\overline{B}.\overline{C}}) + B.C = (\overline{\overline{A}} + \overline{\overline{B}}).\overline{C} + BC = (A + B).\overline{C} \\
 &+ B.C = A.\overline{C} + B.\overline{C} + B.C = A.\overline{C} + B.(\overline{C}+C) = A.\overline{C} + B
 \end{aligned}$$

SOP vs. POS

When manipulating an expression, it is often useful to leave the expression in a standard form:

- $A B + \bar{A} C + \bar{C}$ is a sum of products (SOP)
- $(A + B) (\bar{C} + \bar{B})$ is a product of sums (POS)

Both forms are equivalent, but it is likely that the final version is in one of the standard forms, most likely SOP

Logic Gates

- Computer hardware is made using logic gates
- There are three basic logic gates: AND, OR, NOT and the derived negated gates: NAND, NOR, XOR, XNOR.
- All logic systems may be built out of the first three, or out of NAND and NOR gates.
- These gates are implemented in standard configurations, and later on these in turn are combined to produce the circuits.
- AND gate: Two inputs and one output. The output is true (1) only when all inputs are true (1)
- The equivalent Boolean expression is: $A \cdot B$

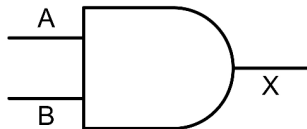


Figure: AND gate

AND Gate

The behaviour of a gate may be expressed using a truth table, defining the output for all possible combinations of input

Table: A truth table for $A \cdot B$

Row	Inputs		Output
	A	B	$A \cdot B$
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

OR Gate

The output of the gate is true when ANY one of the inputs is true: $A + B$

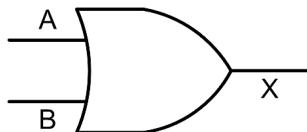


Figure: OR gate

The truth table corresponding to the OR gate is:

	Inputs		Output
Row	A	B	$A + B$
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

NOT Gate

The output is the opposite state to the input, i.e. the output is the input inverted

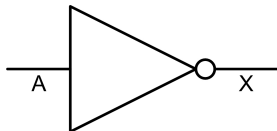


Figure: NOT gate

Table: NOT gate truth table

	Input	Output
Row	A	\overline{A}
0	0	1
1	1	0

Logic

- Combinational logic consists of a logic circuit built with logic gates
- Implements an output determined by the logic gates and the current set of inputs
- There is no memory in the system, that is, the output depends exclusively of the inputs
- An example

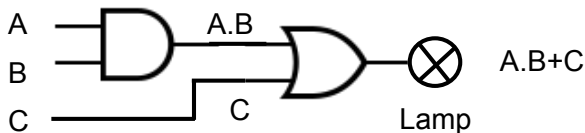


Figure: An example of combinational logic

Logic

- Build a circuit from a given Boolean expression
- This is the reason why we try to simplify expressions, so circuits are easier and cheaper to build
- Consider for example, the SOP expression $A.B + C.D$
- This is built very simply with two AND gates joined together with an OR gate

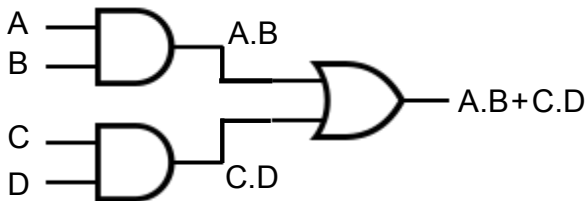


Figure: A simple combinational circuit

Student Exercises

Draw logic circuits for the following Boolean expressions:

1 $A.B + \overline{C} + D$

2 $A + B.C.D$

3 $A.B + \overline{C}.D$

Universal Logic Gates: NAND

- The universal logic gates, NAND and NOR are used Instead of the basic AND, OR, NOT as they are more flexible in that *only one type of gate* is needed
- NAND gate: It is equivalent to: $X = \overline{A \cdot B}$ (NOT(A AND B))
- The symbol for the NAND gate is:

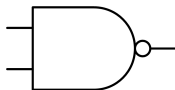


Figure: The NAND gate

- Another notation, also widely used, is:

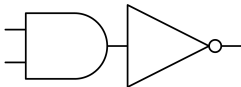


Figure: Another symbol for a NAND gate

NAND Gate

The truth table associated with the NAND gate is:

Table: NAND gate truth table

	Inputs		Output
Row	A	B	$\overline{A \cdot B}$
0	0	0	1
1	0	1	1
2	1	0	1
3	1	1	0

NOR Gate

- NOR gate: It is equivalent to the Boolean expression $X = \overline{A + B}$ (NOT(A OR B))
- The symbol for the NOR gate is shown:

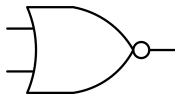


Figure: The NOR gate

- An equivalent symbol is:

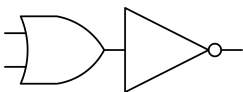


Figure: Another symbol for a NOR gate

NOR Gate

- NOR gate: It is equivalent to the Boolean expression $X = \overline{A + B}$ (NOT(A OR B)).
- The symbol for the NOR gate is:

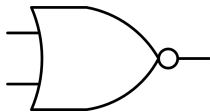


Figure: The NOR gate

- An equivalent symbol is:

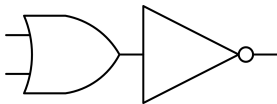


Figure: Another symbol for a NOR gate

NOR Gate

The truth table associated with the NOR gate is:

Table: NOR gate truth table

	Inputs		Output
Row	A	B	$\overline{A + B}$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	0

XOR gate

- XOR gate: Very commonly used in digital logic, Exclusive OR: $A \oplus B$
- The output is true when the inputs are different, and it is false when the inputs are the same



Figure: The XOR gate

- The truth table associated with the XOR gate is:

Table: XOR gate truth table

	Inputs		Output
Row	A	B	$A \oplus B$
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

XNOR gate

- XNOR gate: the negation of the XOR gate; corresponds to the Boolean expression $\overline{A \oplus B}$

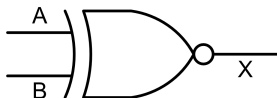


Figure: The XNOR gate

- The truth table associated with the XNOR gate is:

Table: XNOR gate truth table

Row	Inputs		Output
	A	B	$\overline{A \oplus B}$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

Chapter 8

Error Correcting Codes-Digital Logic

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Detect and correct 1-bit transmission errors, using simple parity and Hamming codes
- Design and simplify simple digital systems
- Describe some standard digital components, and their functionality

Errors

- When transferring data within a computer errors can occur
- These errors may cause some 1s to become 0s and some 0s to become 1s, as in "ABC" = 1000001 1000010 1000011, may become 1000001 1000010 1001011 = "ABK"
- In data communications if an error is detected it is easy to fix by requesting retransmission, but when writing data to disk that is not possible
- Extra *redundant bits* (also called *check or parity bits*) are added to the original information before transmission
- Let's add an extra *parity bit* to a 7-bit ASCII character to detect an error
- Depending on the number of 1s that we choose, two different parities, even and odd, may be used
- There are no advantages of one over the other, so they are used indistinctly

Parity

- Even Parity: for ASCII characters bit 7 (that is, the eighth-bit, the MSB) is set so that the total number of 1s in a character is even, as in:

'A' = 100 0001 Parity = 0 Code = 0100 0001

Parity Bit = 0

'C' = 100 0011 Parity = 1 Code = 1100 0011

Parity Bit = 1

- Odd Parity: the parity bit is set so that the total number of 1s is odd, as in:

'A' = 100 0001 Parity = 1 Code = 1100 0001

Parity bit = 1

- Any 1-bit error will change the number of 1s and 0s in the symbol, and hence the parity of the symbol

NOTE: A single parity bit can only detect an error, it cannot correct errors

Hamming Codes

Consider this:

- 1 We are expecting a message which is either 0 or 1
- 2 We receive 1
- 3 If there is a one-bit error, we wouldn't know it
- 4 We cannot tell whether the message is right or wrong

Hamming Codes

Now, we add a parity bit to the messages, and we say "there must be an even number of 1s (either 0 or 2)", so the messages are 00 and 11:

- 1 We are expecting a message which is either 00 or 11
- 2 We receive 01
- 3 We know now that there has been an error in transmission
- 4 We cannot tell which bit is wrong
- 5 We don't know whether the original message was 00 or 11

We are able to detect the error, but we cannot correct it

Hamming Codes

- Parity schemes are an example of a *Hamming Codes*
- Consider now:
 - 1 We are expecting one of only 2 symbols: 000 and 111
 - 2 We receive 001
 - 3 If we assume that there is only a one-bit error, what was the message sent?
- Obviously 000, because 111 has more than one bit difference with 001
- Since there are 3 bits difference between 000 and 111, we can find the mistake and correct it

Error Correction

- An *error correcting code* can detect and correct bit errors in a (binary) code
- In 7-bit ASCII, a 1-bit change to a symbol gives another valid symbol
- It is then not possible to detect or correct a one-bit error
- To detect or correct mistakes, codes use extra bits to increase the distance between valid symbols, as in:

1111 1000	->	111 1001	->	1111 1011	->	1111 1111
valid		invalid		invalid		valid

- In a code, the number of bit changes between two valid symbols of the code is called *Hamming Distance*
- In the example, the Hamming Distance may be made constantly 3, so 3 bits must change to arrive at the next valid symbol
- If one bit error is detected, it may be corrected by selecting the closest valid symbol

Error Correction

- To detect up to K bit errors, the (minimum) Hamming distance between two valid code words must be $K+1$
- Simple parity makes 2 the distance between ASCII symbols
- To correct up to K bit errors, the Hamming distance between two valid code words must be $\geq 2K+1$
- To detect and correct 1 error requires a Hamming distance of 3

Hamming Code

- In the general case, the way we insert those bits is up to the scheme used
- The most popular one is the Hamming code
- In the Hamming code the number of Parity/Check bits p must be:

$$2^p \geq m + p + 1$$

here m = number of data bits and p = number of check bits

Example: Find the number of check bits required to detect and correct a single bit error in the BCD code for $9_{10} = 1001_2$

- If we try $p = 2$ then $2^p = 2^2 = 4$, but since $4 + 2 + 1 = 7$ and 4 is not ≥ 7 , $p = 2$ is not enough
- If we now try $p = 3$ then $2^p = 2^3 = 8$ and $m + p + 1 = 4 + 3 + 1 = 8$.
Therefore $p = 3$ is sufficient

Example

- A Hamming code includes several parity bits
- In principle the parity bits may go in many different places, but the most common use is interspersed with the data bits
- To get the position of the check bits we count the bits from right to left (i.e. LSB to MSB) starting from 1, and place the check bits at positions 1, 2, 4, 8, 16, 32, etc, all powers of 2
- All the other bits are data bits, part of the message

Table: Hamming code for 6 data bits, P = parity bit, D = data bit

Bit position	10	9	8	7	6	5	4	3	2	1
Data/Parity	D6	D5	P4	D4	D3	D2	P3	D1	P2	P1

Hamming Code

- P1 covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.
P1 covers positions where there is a 1 in the first binary bit of the position number binary expression (highlighted in boldface): 000**1**, 001**1**, 010**1**, ...
- P2 covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.
P2 covers positions where there is 1 in the second binary bit of the position number binary expression: 00**1**0, 00**1**1, 01**1**0, 00**1**1, ...
- P3 covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.
P3 covers positions where there is a 1 in the third binary bit of the position number binary expression: 0**1**00, 0**1**01, 0**1**10, 0**1**11, ...
- P4 covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.
P4 covers positions where there is a 1 in the fourth binary bit of the position number expression: **1**000, **1**001, **1**010, **1**011, **1**100, **1**101, **1**110, **1**111

Hamming Code

We can think about it in a different way:

- P1 covers bits 1, 3, 5, 7, 9, 11, 13, 15, etc.
P1 covers one bit, skips one bit
- P2 covers bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.
P2 covers two bits, skips two bits
- P3 covers bits 4, 5, 6, 7, 12, 13, 14, 15, etc.
P3 covers four bits, skips four bits
- P4 covers bits 8, 9, 10, 11, 12, 13, 14, 15, etc.
P4 covers eight bits, skips eight bits

Example

Determine the single bit error-correcting Hamming code for the data 1011_2 . Use EVEN parity. From the formula we can see that $p = 3$ is sufficient:

7	6	5	4	3	2	1
1	0	1		1		

- P1 checks bits 1, 3, 5, 7 = * 1 1 1, P1=1
- P2 checks bits 2, 3, 6, 7 = * 1 0 1, P2=0
- P3 checks bits 4, 5, 6, 7 = * 1 0 1, P3=0
- Therefore the final code is 1 0 1 0 1 0 1

7	6	5	4	3	2	1
1	0	1	0	1	0	1

Hamming Code

- The idea is that the parity bits cover each bit (data and parity) with a unique pattern
- Each bit (including the parity bits) is covered by a unique combination of parity bits
- For example, bit 5 is covered by bits P1 and P3 only and exclusively, and bit 4 is covered by P3 exclusively
- If we find a parity error on bits P1 and P3, but no other parity errors, we know that the problem is with bit 5
- If bit 5 is a 1, we change it to 0, and if it is a 0 we change to 1

Multiple Errors

- We are only interested in 1-bit errors
- In computer operation 2-bit errors are very unlikely
- If the probability of a 1-bit error is of the order of 10^{-9} , that is $1/1,000,000,000$, this is really small, but if a computer makes 10,000,000 moves a second, on average you get an error every 100 seconds = less than 2 minutes
- 2-bit errors occur when there is one error on two lines at the same time. That is, the probability is the product of the two probabilities 10^{-18} . With the same computer you get a 2-bit error once every 10^{11} seconds = once every 3,171 years
- With data communications it is quite common to have noisy lines, so the probability of errors increases dramatically
- It often happens that there is a short period when may be multiple-bit errors
- Other more appropriate schemes are used instead

SECDED Coding

- Changes of two or more bits either are not going to be detected by a Hamming scheme
- *SECDED Coding* (single-error correction, double-error code) makes possible the correction of single-bit errors and the detection, but not correction, of double-bit errors
- We use the unused P0 bit as a parity (odd or even) bit for all the bits in the transmitted symbol
- If one bit is in error, the Hamming checks will fail for a particular set of bits, and also the overall parity will be wrong
- We may then assume that there has been a 1-bit error and the Hamming code can correct it
- If 2 bits are in error, some Hamming checks will fail but the overall parity will be right, so more than 1 bit was in error but we are not able to correct it

Summarising SECDED Codes

Summarising, using SECDED coding:

- If some Hamming checks fail, and there is also an overall parity error, we assume that there was a 1-bit error and we correct it as before
- If some Hamming checks fail, and there is no overall parity error, we assume that there have been 2 or more errors, but we don't correct them

Of course, if there have actually been more than 2 errors, the SECDED scheme will give a wrong result

Designing Digital Logic Systems

- A digital logic system is the translation of a logic expression to a circuit that can implement that logic
- It is written first as a Boolean expression or a truth table that provides the Boolean expression
- From the Truth table the Boolean expression can be written and then simplified
- The simplification is important because it reduces the size and complexity of the implemented circuit
- The logic circuit is then drawn from the simplified Boolean expression

Designing Digital Logic Systems

Example: The output is true only when the input is less than 2 or greater than 5

Table: Truth Table for the example

Row	A	B	C	X	Term
0	0	0	0	1	$\overline{A} \overline{B} \overline{C}$
1	0	0	1	1	$\overline{A} \overline{B} C$
2	0	1	0	0	
3	0	1	1	0	
4	1	0	0	0	
5	1	0	1	0	
6	1	1	0	1	$A B \overline{C}$
7	1	1	1	1	$A B C$

The resulting Sum of Products is: $\overline{A} \overline{B} \overline{C} + \overline{A} \overline{B} C + A B \overline{C} + A B C$

Half Adders

To add binary numbers, we need to add individual bits properly, we need to calculate their sum and the corresponding carry

The Truth Table for a 1-bit adder — a half-adder — is shown:

Table: Truth Table for a Half-Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adders

The figure depicts the logic for a half-adder:

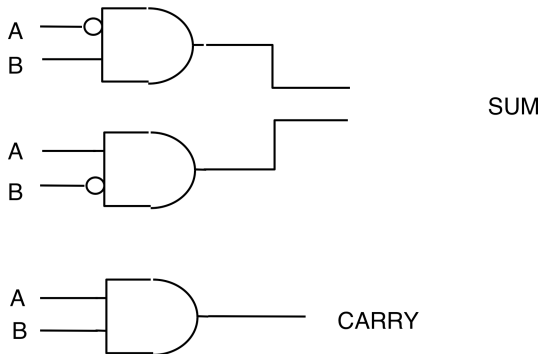


Figure: Half-adder digital logic

$$\text{Sum} = \bar{A}.B + A.\bar{B}, \text{ Carry} = A.B$$

Full Adder

- To add two bits properly we have to make sure that we add the carry properly
- The design could be carried out again via a truth table, but a better method is to use two half-adders, since this approach is based on existing components
- A full-adder is able to add in the carry from the previous column in a two-digit binary number

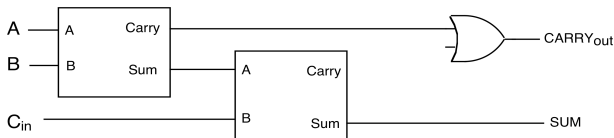


Figure: Full-adder digital logic

Multidigit Adders

- A multi-digit adder may be implemented by cascading full-adders
- The figure shows 4 full-adders, in which the carry out of each full-adder is fed into the next one
- The scheme is initialised with 0 as the first carry in, and the last carry out is the carry out of the whole adder

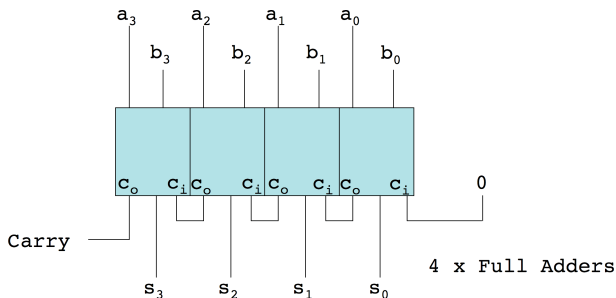


Figure: Parallel adder digital logic

Integration of Components

The design of a function specification must be translated to a combination of digital components. For reasons of cost, these components should be standard, and arranged together in integrated circuits of increasing complexity and size:

- small scale integration, SSI: 1 to 10 components (e.g. logic gates)
- medium scale integration, MSI: 10 to 100 components (e.g. adders, shift registers)
- large scale integration, LSI: 100 to 100,000 components (e.g. arithmetic-logic unit)
- very large scale integration: 100,000+ components (e.g. complete microprocessors)

Encoders

- An encoder converts 'one-of-many' inputs into a binary coding
- Typically there are 2^n inputs and n outputs, such as $2^3 = 8$ inputs and a 3-bit output
- It can translate a line put high by a device seeking to interrupt the processor to the corresponding binary number that is stored as the bit mask of the status vector. The output binary coding represents the number of the input that is active

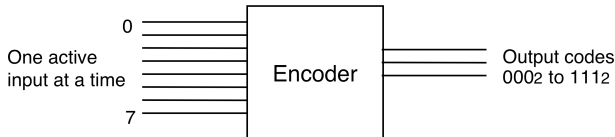


Figure: A 3-bit encoder

Decoders

A decoder is the opposite to an encoder, where for n inputs there are 2^n outputs

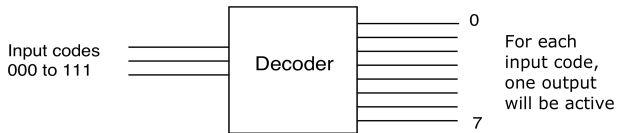


Figure: A 3-bit decoder

Multiplexor

A multiplexor chooses one input to be connected to a single output. This selects a data source to send to a destination, according to the value of the selector

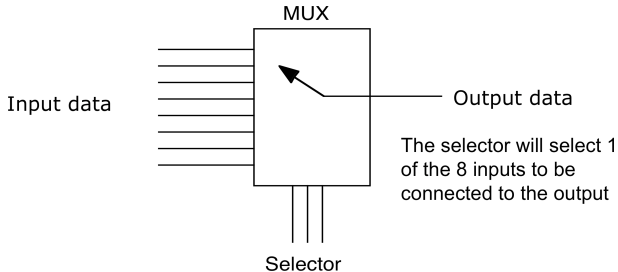


Figure: A 3-bit multiplexor

Demultiplexor

A demultiplexor directs a single input to one of a number of outputs, according to the value of a selector

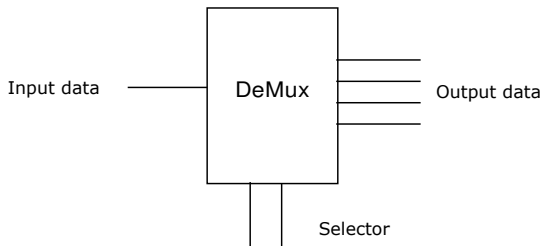


Figure: A 2-bit demultiplexor

Chapter 9

Assembly Language

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Develop more advanced assembly language programs with further addressing modes, to implement loops
- Describe the structure and function of the 68K stack

68K Addressing Modes

Up until this point, we have used several addressing modes suitable for the problems we faced:

- Absolute `$3000` or a `label`
- Register direct `dn` or `an`
- Immediate `#`
- Address Register indirect (`an`)

There are more addressing modes available for the programmer, which offer further functionality: `(an)+`.

Address Register Indirect with Post Increment

In the Address Register indirect with post increment (`an`)+, the address register is automatically incremented after use, as in:

```
move.b    (a4)+, d3
```

This auto increment saves having to use an extra instruction to modify the address register after access, as we had to do in previous programs

Address Register Indirect with Post Increment

```

;inputs a string of characters from the keyboard into memory
    org        $1000
    move.l     #$7ffe,sp        ;initialise stack pointer
    move.l     #string,a6      ;initialise string pointer
next  move.b    #5,d0           ;get ready to get a char
    trap       #15             ;go get it
    cmp.b      #cr,d1          ;is it a return?
    beq        finish          ;all done if a cr
    move.b     d1,(a6)+         ;put in a6 and increment
    bra        next
finish move.b    #9,d0          ;prepare to go back
    trap       #15             ;go back

data   org        $1500        ;data starts here
string ds.b      100           ;space for 100 characters
cr      equ       $0d           ;define cr
end

```

Address Register Indirect with Post Increment

Exercise: Modify and test the program above to include an echo of the input to the display

With this new addressing mode it easier to program loops in assembly. Let's write an example that adds three bytes of data

Address Register Indirect with Post Increment

```

;add three bytes of data starting at $1200
start      org          $1000
           move.l       #$7ffe,sp          ;initialise sp
           move.l       #data,a1          ;init address register
           move.b       #3,d2             ;initialise counter = 3
           clr.b        d4                ;clear result register
loop       add.b        (a1)+,d4          ;get data, add, incr a1
           sub.b        #1,d2             ;decrement counter
           bne          loop              ;do all the adds
           move.b       d0,(a1)           ;save result in memory
           move.b       #9,d0             ;go to OS
           trap         #15

           org          $1200
data       dc.b         4                 ;data space for
           dc.b         12                ;three data items
           dc.b         7
           ds.b         1                 ;space for result
           end

```

Address Register Indirect with Post Increment

- Loops in high-level languages are translated by compilers to assembly loops such as the one above
- The previous example is the assembly equivalent of:

```
result = 0;

for (i = 3; i > 0 ; i--)
{
    result += data[i];
}
```

- Exercise: Draw a diagram in memory of the location in memory of this program, including instructions and data. Trace the program first by hand, and then using SIM68K

Address Register Indirect with Pre Decrement

- Similarly to the previous addressing mode, the Address Register Indirect with Pre-Decrement may be used to:
 - decrement the address register by the required amount (b, w, l)
 - perform the operation at the address indicated
 - For example:

```
move.w    #$A4, -sp
```

- This is particularly useful when managing the stack

NOTE: there are no pre-increment or post-decrement equivalent instructions

Address Register Indirect with Displacement

- A further addressing mode is convenient when dealing with structures such as the stack: Address Register Indirect with Displacement $d(an)$
- For example, if we write:

```
move.b    4(a4),d0
```

the effective address that goes to the address bus is

$$d + an = 4 + a4$$

- This addressing mode is most often used to access consecutive data, such as an array or string

Reading a String

;reads and displays the 7th character of a string.
;Test in the lab and observe d0 in the register dump
;when the program terminates.

```
start      org          $1000
           move.l       #$7ffe,sp      ;init stack
           move.l       #string,a0     ;init pointer
           move.b       6(a0),d1       ;get char, (7-1)
           move.b       #6,d0          ;send char
           trap         #15            ;to display
           move.b       #9,d0          ;go back
           trap         #15
```

; This part is for the data.

```
data       org          $1100
string     dc.b         'Hello again!'
           end
```

Counting Characters

;counts and displays the number of characters in a string.
;Fill in the missing branch instruction and the label for the loop
;Test in the lab. Fix any bugs or omissions. Limitations?

```
start      org          $1000
           move.l       #$7ffe,sp          ;init stack
           clr.b        d1                 ;zero the counter
           move.l       #str,a5            ;init string pointer
           move.b       (a5)+,d2           ;get char from string
           cmp.b        #null,d2          ;see if end of string
           beq          done              ;done, if null
           add.b        #1,d1              ;bump count
           b..          ...               ;loop till end
done        add.b        #$30,d1           ;binary to ASCII
           move.b       #6,d0             ;display count
           trap         #15               ;exit
data        org          $1050
str         dc.b        'count me',null
null        equ         0
end
```

Printing a String

The previous code implements a very common logic, a `while` loop to print the number of characters in a string, as in the following pseudocode:

```
i = 0;
while (str[i] is not NULL)
{
    i++;
}
println(i);
```

Sample Program Exercise

Write a similar assembly language program to implement the following `while` loop:

```
i = 0;
while (str[i] is not NULL)
{
    print(str[i]);
    i++;
}
println();
println(i);
```

The 68K Stack

Recall that a generic stack is a LIFO structure:

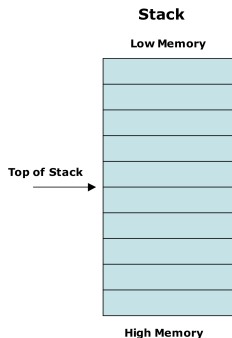


Figure: A stack

In the 68K assembly, the stack is set aside at the beginning of the code:

```
move.l    #$7ffe,sp
```

The 68K Stack

- The default stack pointer `sp` is register `a7`
- Other address registers may be also used as stack pointers, but `a7` is specifically designed to work properly with the stack
- The `sp` moves in increments of 2, so it is always pointing to an even address and therefore it is properly aligned in memory
- The 68000 stack can be seen as 16 bits wide, although bytes, words and long words can all be stored

The 68K Stack

- To store data on the stack we need to implement push and pop instructions
- Consider the following sequence of push and pop instructions:

```
move.b    d0,-(sp)      ; d0 = 9A
move.b    d5,-(sp)      ; d5 = 35
move.b    d3,-(sp)      ; d3 = F7
move.b    (sp)+,d3
move.b    (sp)+,d5
move.b    (sp)+,d0
```

The 68K Stack

- This sequence of instructions results in the sequence of stack states depicted in the figure:

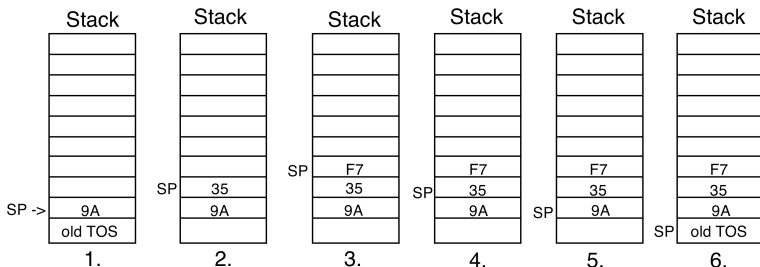


Figure: Stack diagram corresponding to sequence

- As data is popped off, the stack space previously used is available for more data
- That region part of memory is recycled, the data put on the stack remains there, but it will be over-written when new data is pushed onto the stack

The 68K Stack

- As a consequence of the required memory alignment, the stack pointer is always pointing to an even memory address
- A byte only uses a byte size location and leaves a byte size location empty. Although this is a loss of space, it is really negligible with large modern memories

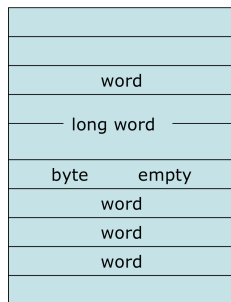


Figure: Space allocation in the 68K stack

Chapter 10

Digital Logic Assembly Language

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After studying this chapter, students should be able to:

- Describe standard memory elements such as Flip-Flops
- Explain how standard elements may be used to build registers, counters and dividers
- Explain the notion of subroutines and the related instructions `bsr` and `rts`
- Describe the characteristics of the 68K stack and its use to pass parameters
- Explain how subroutines may retrieve parameters from the stack

As discussed before, there are two kinds of memory elements in digital logic systems:

- Flip-Flops (used in static memory)
- Capacitors (used in dynamic memory)

The SR Flip-Flop

- The most common Flip-Flop is the SR flip-flop
- The truth table of the SR FF is given below

Table: SR Flip-Flop truth table

<i>Set</i>	<i>Reset</i>	Q
0	0	No change
0	1	0
1	0	1
1	1	Undefined

Computer Clock

- All computers use a clock to coordinate computer operations
- To properly sequence operations
- Making sure that circuits are in a stable, correct state before using their outputs
- A clock is a hardware device that produces a very regular, square wave:



Figure: A clock wave

- If we synchronise all activity with clock cycles, we give circuits enough time to change state and be stable before the next step
- The time between two rising edges is called the *clock cycle time* measured in Hertz ($1\text{Hz} = 1 \text{ cycle per second}$)
- Typically a modern processor has a clock cycle of 1, 2, 3 Giga Hz

Clocked Flip-Flops

- Flip-Flops may be coordinated by the use of a clock to control their downstream logic flow
- With FFs, the time that takes for the logic to flow through makes them difficult to use in circuits
- *Clocked Flip-Flops* use extra logic to make them more useable in logic systems
- Two common types of Clocked FFs are the D and JK FFs:



Figure: The D and JK clocked FFs

- Clocked FFs gate their inputs to the output with a clock signal

The D Flip-Flop

- The operation of a D FF is quite simple: the logic level on the D input is transferred to the output Q when the clock tick occurs

Table: The D Flip-Flop truth table

D	Q_{n+1}
0	0
1	1

- This is used often to transfer binary information in synch with the clock
- For example in computer registers, counters and shift registers to store multi-digit binary data

The JK Flip-Flop

- JK FFs are more complex, but more flexible
- This is the truth table of the JK FF:

Table: The JK Flip-Flop truth table

J	K	Q_{n+1}	Result
0	0	Q_n	no change
0	1	0	reset
1	0	1	set
1	1	$\overline{Q_n}$	toggle

Registers

- A register, such as a CPU register, is a group of FFs arranged to receive and store multi-digit binary data
- The input data is stored in the register on the clock.
- The stored data can be read at the outputs at the next tick of the clock.
- In this way, the clock synchronises transfers between registers and between registers and memory
- Static memory works like this, using D FFs

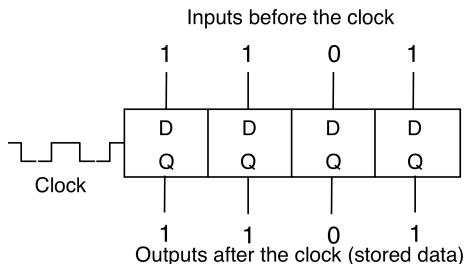


Figure: A register made out of D FFs

Shift Registers

- Shift registers are organised so that on each clock input, the data stored in the register moves one position to the right
- This is used to translate parallel data into serial data
- This is known as *serial data transfer*

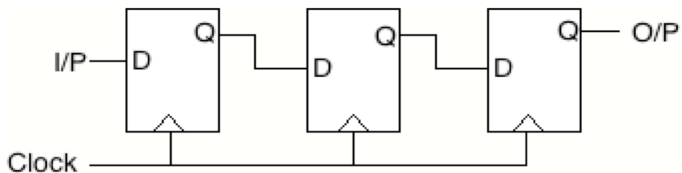
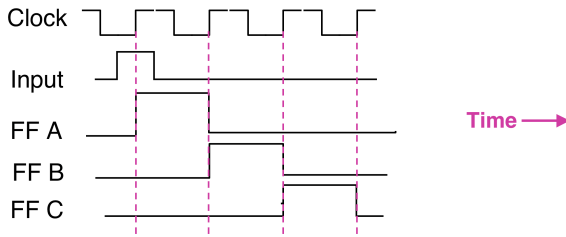
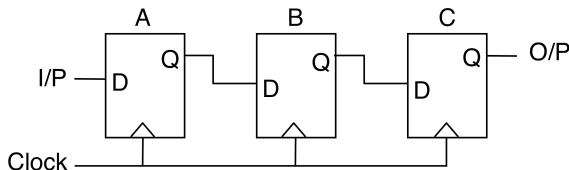


Figure: A shift register

Shift Registers

- If the Flip-Flops are clocked on the positive edge of the clock, every time the clock wave goes up the FFs advance the bits
- A register with a HIGH input signal for a brief period of time (i.e. the input of a 1)



Dividers

- This device consists of 3 JK Flip-Flops, where the first FF clock input is connected to the clock, but each following FF clock input is connected to the \overline{Q} output of the previous FF.
- All FFs have both J and K inputs 1 (HIGH), so each time an input is received in the clock, the FF toggles.
- Assume that the FFs are clocked on the positive edge of the clock, and that all Q outputs are initially zero

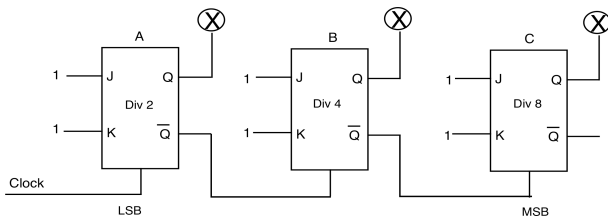


Figure: A signal divider

Dividers

- The first FF toggling each time the clock goes high
- The second FF goes high on the positive edge of the output of the first FF, that is, when the \overline{Q} output of the first goes from 0 to 1, *and only then*
- When the second positive edge of the clock is input into the first FF, the FF toggles again, and as a consequence \overline{Q} goes from 0 to 1, and this time the second FF toggles Q from 0 to 1.
- The second FF only toggles each second time the first one does, and then the second FF k.pdf the 0 and 1 states double the time the first one does.

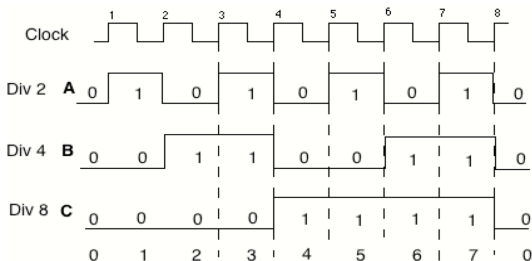


Figure: A divided wave

Counters

- If we observe the output of all the FFs rather than each individually, we can see that the output of the 3 FFs form a binary increasing sequence, if we consider the first FF as the LSB:

000

001

010

011

100

101

110

111

- We have an 'up counting' binary sequence
- If we look at the \overline{Q} outputs, we obtain a 'down counting' sequence

Subroutines

- Subroutines are used so that only a single copy of a frequently used piece of code is required
- Save programmer time by avoiding having to rewrite the same code all the time
- Save memory space because there is only one copy; a routine that uses 100 bytes of memory and is used 1000 times would take 100,000 bytes of memory instead of only 100 bytes

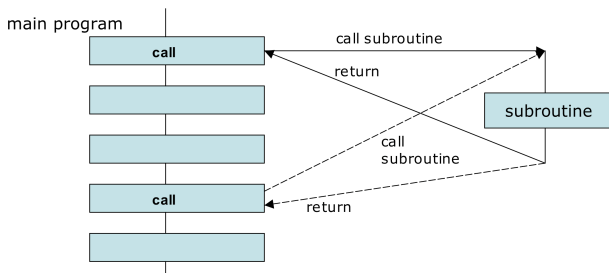


Figure: Main program calling a subroutine twice

Subroutine Calls

- When the main program finds a subroutine call, it suspends execution and calls the subroutine
- While the subroutine is executing the main program is suspended
- At the end of its execution, the subroutine returns control to the main program, which then resumes execution
- This process is repeated each time the subroutine is called

Subroutine Calls

- Programs call subroutines using `bsr` (branch to subroutine)
- Program use `rts` (return from subroutine) to return to the caller
- `bsr` does two things:
 - pushes the return address onto the stack; the return address is the address of the instruction after the `bsr` instruction
 - puts the address of the subroutine into the PC
- The `rts` instruction pops the return address from the stack into the PC; this restores execution to its original position

Subroutine Calls

Say a program is executing and it finds a `bsr` instruction at location `$23AB`:

- 1 the address of the *next* instruction (say `$23B0`) is pushed on the stack
- 2 the starting address of the subroutine is loaded into the PC

When the `rts` instruction is reached the address `$23B0` is popped off the stack into the PC

The following code illustrates the use of subroutines to store keyboard input into a string

Program With Subroutines

;inputs keys into a string using subroutines.Test in lab.

```

code      org          $1000          ;code space
          move.l       #$7ffe,sp      ;stack at the top memory
          move.l       #keys,a2       ;string pointer
next      bsr          input          ;get a key
          cmp.b        #cr,d1         ;is this the end of input
          beq          exit           ;exit on cr
          bsr          output         ;print the key
          move.b       d1,(a2)+       ;put key into string
          bra          next           ;loop till cr
exit      move.b       #9,d0          ;exit
          trap         #15

```

Program With Subroutines

```
input    move.b    #5,d0          ;subroutine to read a key
         trap      #15
         rts        ;return to caller
output   move.b    #6,d0          ;subroutine to display key
         trap      #15
         rts        ;return to caller
keys     ds.b      100            ;reserve 100 bytes
cr        equ      $0d
end
```

Reentrant Code

If more than one subroutine modify the contents of d7, the contents of this register may depend on the order in which the subroutines are called. The code is *not reentrant*.

; Suggested modification. Modify!!

.....

.....

```
input      ????                ;push d0 on the stack
          move.b      #5,d0    ;subroutine to read a key
          trap        #15
          ????                ;restore d7
          rts                ;return to caller
```

```
output     ????                ; push d0 on the stack
          move.b      #6,d0    ;subroutine to display key
          trap        #15
          ????                ;restore d7
          rts                ;return to caller
```

.....

.....

Exercises

For the following problems, draw flow charts, write the corresponding programs and test them. Use subroutines where possible.

- 1 Modify the program above to make it re-entrant as shown in the suggested modification
- 2 Add 5 words of data in memory and save result
- 3 Print a message on the display and then input the user response. Display the response and store it in memory. Examine the string in memory with MD
- 4 Consider how the use of the string input/output functions provided by EASy68K could make this easier

Parameters

- It is often necessary to pass information to a called subroutine
- These pieces of data are known as *parameters* or *arguments*
- This technique is known as *parameter passing*
- Parameters may be passed by three different methods:
 - 1 In registers
 - 2 In memory
 - 3 On the stack

Parameter Passing

- Passing parameters on the stack is preferred, since it has virtually no space limitations and is a 'fully re-entrant' method
- Parameters are pushed on the stack and then the routine is called:

```
move.w    #6,-(sp)    ;push parameter 1
move.w    data,-(sp)  ;push parameter 2
bsr       xyz         ;subroutine call
```

- The subroutine may access the parameters, but carefully since the bsr instruction has pushed the program counter PC on the stack

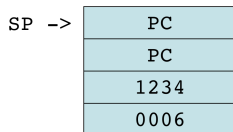


Figure: Parameter allocation in the 68K stack

Parameter Passing

- Both parameters are integer numbers, but they have been passed on the stack in different ways:
 - parameter 1 is an integer, immediately passed on the stack in the instruction itself, so the *value* of the parameter is pushed
 - parameter 2 is also an integer, but it has been passed via its address in memory, that is, the *location* of the data is copied on the stack
- The PC return address is 2 words, and each of the two parameters (in this case) are 1 word
- The subroutine needs to know the order in which the parameters have been pushed and their type, so it can retrieve them properly

Parameter Passing

- The subroutine can use the addressing mode `d(An)` to access each word on the stack:

```

move.w    4(sp),d5           ;get parameter 2
move.w    6(sp),d6           ;get parameter 1

```

- The value of the PC has been skipped when retrieving the parameters
- The status of the stack is (note the space for the result):

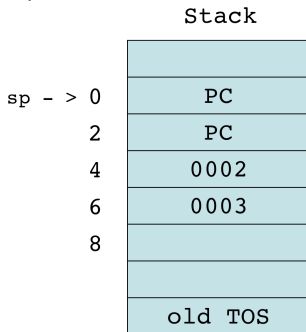


Figure: Passing parameters on the 68K stack

Parameter Passing

; Two parameters to a subroutine, answer returned on the stack

```
start      org          $1000
           move.l       #$7ffe,sp      ;init stack pointer
           sub.l        #4,sp          ;stack space for answer
           move.w       #3,-(sp)       ;push parameter 1
           move.w       #2,-(sp)       ;push parameter 2
           bsr          prod           ;go to multiply
           add.l        #4,sp          ;remove parameters
           move.l       (sp)+,ans       ;pop result into ans
           move.b       #9,d0          ;ready to exit
           trap         #15
```

```
prod       move.w       4(sp),d0        ;get parameter 2
           move.w       6(sp),d1        ;get parameter 1
           mulu         d0,d1           ;multiply
           move.l       d1,8(sp)        ;result on the stack
           rts          ;return to caller
```

```
ans        ds.l        1               ;space for result
```

```
end
```

Chapter 11

Assembly Language

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After completing this chapter, students should be able to:

- Develop more complex assembly language programs with advanced addressing modes, subroutines and parameter passing

A Large Project

You need to write an assembly language program that will meet the following requirements:

- Requirement 1
 - Allow the user to enter a string of up to 100 characters
 - You must echo each character to the screen and store it in memory as it is typed
 - At the end of the user will press Enter, (INCH returns `cr` in this case)
 - You must terminate the string in memory with a `null` character
- Requirement 2
 - After the input, you must loop through the string stored in memory and count the number of words in the string. A word is defined as any series of alphanumeric characters (a–z, A–Z and 0–9).
 - E.g. 'Fred can, run!?!' contains 3 words ('Fred', 'can' and 'run').
 - E.g. 'What's that' also contains 3 words ('What', 's' and 'that').
 - E.g. '\$AU 123.466' also contains 3 words ('AU', '123' and '466').

A Large Project

- Requirement 3

- You are also required to keep the position of the first character of longest word in the string
- If multiple words are of the same length, you must keep the position of the *first occurrence* of a word of that length.

E.g. 'This is a short word' must maintain a pointer to 'short'.

E.g. 'A long sentence contains long words' must maintain the first position of 'sentence'.

Requirement 4

- Once the loop has been completed you must display:
 - the number of words (in multi-digit *hexadecimal*)
 - the longest word on the screen with an appropriate message, as in:

There were 3 words in the string
The longest word was: fred

Development Strategy

- As the complexity and size of the programs increases it is necessary to tackle the problem in a stepwise fashion
- It is not possible to write all the required code in one go
- This approach requires careful planning, making sure that each step is implemented and fully tested before making further progress
- To illustrate the approach, we are going to suggest a development strategy for the project described above

How Do We Develop This Project

- ❶ Allow the user to enter a string in memory and terminate the program. Test that the string is correctly stored in memory, including the null at the end. This ensures that your input is OK.
- ❷ Loop through the string and display it by displaying each character at a time. This makes sure that your loop through the string works properly.
- ❸ Modify the code to print *the first word*. This makes sure that you are able to print a word properly.
- ❹ Loop through the string and display a word at a time, one word per line. This ensures that your program properly recognises each word.
- ❺ Write a subroutine to display a number in its hexadecimal form. Test in a separate program.
- ❻ Include the subroutine into the code, and count and display the number of words.
- ❼ Modify the code to print the length of each word together with the word one per line. This makes sure that you are getting the length of the words right.
- ❽ Modify the code to keep and print the length of the longest word.
- ❾ Modify the code to print the longest word by keeping a register pointing to the longest word.

Using Constants

```

LINE_BREAK      dc.b    CR,LF,NULL
PROMPT          dc.b    'Please enter up to 100 characters (Enter when
                dc.b    'finished):',NULL
THERE_WERE      dc.b    'There were ',NULL
WORDS           dc.b    ' word/s in the string',NULL
LONGEST_WAS     dc.b    'The longest word was: ',NULL

LONGWORD_LEN    equ     4        ; the length of a longword in bytes
WORD_LEN        equ     2        ; the length of a word in bytes
NULL            equ     $00      ; used to indicate the end of a string
CR              equ     $0D      ; ASCII code for carriage return
LF              equ     $0A      ; ASCII code for line feed
QUIT            equ     9        ; to indicate that control of the
                                ; computer should return to the OS

INCH            equ     5        ; OS to read from the keyboard
OUTCH           equ     6        ; OS to print to the display
OSCALL          equ     15       ; number 'trap' needs to perform OS calls

```

Reading a char

```

;* get_char *****
;  Waits for a character from the keyboard and then places its
;  ASCII value into the lowest byte of d0.
;  Uses d7 non-destructively
;*****

get_char    move.w    d7,-(sp)      ; save d7's original state
            move.b    #INCH,d7     ; tell the OS to read a character
            trap      #OSCALL      ; call the OS
            move.w    (sp)+,d7     ; restore d7's original state
            rts              ; return to caller

```

Printing a char

```

;* print_char *****
; Prints the character to the screen whose ASCII character is
; found in the lowest byte of d0.  Uses d7 non-destructively
;*****

print_char  move.w  d7,-(sp)      ; save d7's original state
            move.b  #OUTCH,d7    ; tell the OS to read a character
            trap    #0SCALL      ; call the OS
            move.w  (sp)+,d7     ; restore d7's original state
            rts                ; return to caller

```

Reading a String

; Reads a sequence of characters, storing them in the location
; specified on the stack, until the Enter key is pressed.

```
get_string    move.l    a2,-(sp)        ; save states of registers
              move.w    d0,-(sp)        ; that will be used
```

```
gs_loop      move.l    10(sp),a2
              bsr       get_char
              cmp.b     #CR,d0
              beq        gs_done
              bsr       print_char
              move.b     d0,(a2)+
              bra        gs_loop
```

```
gs_done      bsr       line_break      ; move to the next line
              move.b     #NULL,(a2)    ; terminate with a NULL
```

```
              move.w     (sp)+,d0      ; restore register states
              move.l     (sp)+,a2
              rts          ; return to caller
```


Printing a String

```
; * Prints a sequence of characters from memory until a NULL
; * character is encountered.
```

```
print_string    move.l    a2,-(sp)    ; save contents of a2
                move.w    d0,-(sp)    ; save part of d0
                move.l    10(sp),a2   ; address of string to be
                                      ; printed into a2

ps_loop         move.b    (a2)+,d0    ; read byte at current address,
                                      ; increment address register
                cmp.b     #NULL,d0    ; is that byte a NULL?
                beq       ps_done     ; finish up if so
                bsr       print_char  ; print the character to screen
                bra       ps_loop     ; repeat as needed

ps_done         move.w    (sp)+,d0    ; restore d0's original state
                move.l    (sp)+,a2    ; restore a2's original state
                rts                ; return to caller
```

Counting Words

```

count_words    move.l    a2,-(sp)
               move.w    d0,-(sp)           ; save the registers
               move.w    d1,-(sp)           ; that will be used
               move.w    d2,-(sp)

               move.l    14(sp),a2          ; a2: current location
               clr.w     d2                 ; d2: running word count

cw_findstart   move.b    (a2)+,d0          ; get the next character
               cmp.b     #NULL,d0          ; break if end of string
               beq        cw_done           ; has been reached
               bsr        is_word_char
               cmp.b     #0,d1
               beq        cw_findstart      ; until a word character

               add.w     #1,d2              ; increment the word count

```

Counting Words

```

cw_findend    move.b  (a2)+,d0          ; get the next character
               cmp.b   #NULL,d0         ; break if end of string
               beq     cw_done          ; has been reached
               bsr     is_word_char
               cmp.b   #0,d1            ; until a non-word
               bne     cw_findend       ; character is found

               bra     cw_findstart

cw_done       move.w   d2,18(sp)        ; save count on stack

               move.w   d2,(sp)+
               move.w   d1,(sp)+        ; restore registers
               move.w   d0,(sp)+
               move.l   a2,(sp)+
               rts                     ; return to caller

```

Chapter 12

Assembly Language

A/Professor George Fernandez
Updates 2009 Don Gingrich

February 19, 2009

Objectives

After completing this chapter, students should be able to:

- Develop more advanced assembly language programs with further addressing modes, subroutines and parameter passing
- Explain the concept of activation records and frame pointer
- Describe the mechanism of nested subroutine calls
- Explain the consequences of the concepts above for variable scope

Passing Parameters

- Parameters are passed on the stack either by pushing the value by pushing the address of the parameter on the stack
- If two `int` variables `x` and `y` have been declared by in a high-level language
`int x, y;`
and that they are now passed as parameters
- The variables have been declared, the compiler allocates room in memory for the ints, and allows the program to refer to them by their names `x` and `y` instead of their locations
- There are two different ways to make these variables available as parameters: *by value*, or *by address* (also called *by reference*)
 - Passing a variable by value pushes the value of the variable on the stack, and then the copy of the value is manipulated by the subroutine.
 - In contrast, passing a variable by reference pushes the address of the variable on the stack

Passing Parameters

Consider the following assembly code segment:

```

        move.w A,-(sp)      ; push value of A on stack
        pea    B            ; push reference to B on stack
        bsr    sub          ; branch to subroutine
sub      move.l 4(sp),a0     ; a0 points to B
        move.w (a0),d0      ; get B into d0
        move.w 8(sp),d1     ; get A off the stack into d1
        .....
        rts

data    org $2000
        dc.b   A   5        ; first variable
        dc.b   B   7        ; second variable

```

Passing Parameters

- The code pushes the value of A onto the stack, and the address of B onto the stack
- The subroutine retrieves the values from the stack, copies them onto registers and operates with the copies
- We do not have on the stack the address of A, the original value of A will always stay the same
- We pushed the address of B onto the stack with the `pea` instruction, we are able to change the original value of B:

```
move.b #9, (a0)      ; change the value of B to 9
```

- A function or method may access the values of the arguments (*passed in*), and in some cases it may change the original value of a variable passed as an argument (in-out parameter)

Passing Parameters by Value

- The value of the variable is pushed on the stack
- The actual value of the parameter is transferred to the function/method
- This is the safest approach because the original value cannot be changed, even by accident
- In some HHLs it is possible to return a complex value (object/struct) to the caller, changing the value:

```
my_record = update_record (my_record); \newline
```

- Not suitable for large amounts of data
- In order to pass a parameter by value through the stack:

```
move.w A,-(sp)    ; A is pushed on the stack
```

Passing Parameters By Reference

- The address of the variable is pushed on the stack
- The address of the parameter may be used by the function/method
- This will be necessary if the parameter is to be changed by the function
- Recommended in the case of large data volume (object/struct/array), since only the address needs to be pushed on the stack, not the whole object/struct
- In order to pass a parameter by reference through the stack, one may use the instruction:

`pea B` ; the address of B is pushed on the stack

An Example

```

; data1^2 + data2^2 into reslt
    org          $1000
    move.l       #$7ffe,sp
    move.l       #reslt,-(sp)
    move.l       data1,-(sp)
    move.l       data2,-(sp)
    bsr         sumsqr
    lea          12(sp),sp
    move.b       #9,d0
    trap         #15

```

```

;main section
;set up the stack pointer
;pass addr. of reslt thru stack
;pass data1 through stack
;pass data2 through stack
;call subroutine
;restores the stack pointer
;go back

```

An Example

```

sumqsr      org          $2000                ;subroutine
            lea           4(sp),a1            ;get starting address of param
            move.l        (a1)+,d0            ;data1 to d0,
            move.l        (a1)+,d1            ;data2 to d1
            move.l        (a1),a0             ;address of reslt goes to a0
            muls          d0,d0               ;d0 = d0 * d0
            muls          d1,d1               ;d1 = d1 * d1
            add.l         d0,d1               ;d1 = d0 + d1
            move.l        d1,(a0)            ;save result to reslt
            rts

data1       org          $3000                ;data section
            dc.l          $200
data2       dc.l          $100
reslt       ds.l          1
            end

```

Analyse Example

```

        org      $1000                ; start
        move.l   #$7ffe,sp            ; initialise stack
        move.l   #$2000,a1            ; address of string in a1

nxt     move.b   #5,d0                 ; INCH
        trap     #15
        move.b   #6,d0                ; OUTCH
        trap     #15
        move.b   d1,(a1)+              ; store char
        cmp.b    #$0d,d1              ; is it <return>?
        beq      ext                  ; exit if it is
        bra      nxt                  ; otherwise next
ext     move.b   #9,d0                ; back to simulator
        trap     #15

        org      $2000                ; start of string
        ds.b     256                  ; room for the chars
        end

```

To Uppercase

```

                                org      $1000          ; start
                                move.l   #$7ffe,sp       ; initialise stack
nxt                             move.b   #5,d0          ; INCH
                                trap      #15
                                cmp.b    #$0d,d1        ; is it <return>?
                                beq       ext            ; exit if it is
                                cmpi.b   #'a',d0        ; if <'a' not lowercase
                                blt       display        ; so display
                                cmpi.b   #'z',d0        ; if >'z' not lowercase
                                bgt       display        ; so display

                                subi.b    #32,d1         ; to upper

display  move.b   #6,d0          ; OUTCH
          trap    #15
          bra     nxt           ; otherwise next
ext        move.b   #9,d0        ; go back
          trap     #15
          end

```

Stack Frames

- The stack is used dynamically allocate storage procedure data
- We know how room is allocated on the stack when a procedure is called, and how that space must be deallocated when it returns
- The stack should be left as it was before the procedure call
- The block built on the stack is called the *stack frame* or *activation record*
- The stack frame for a procedure is built by the compiler to contain all the information to run the procedure:
 - parameters
 - local variables
 - return value
 - frame pointer fp (more on this below)

Stack Frames

- The specification of a standard calling convention makes possible the use of procedure libraries by defining the structure of the stack frame uniformly for all procedure calls
- Compilers that follow the calling convention generate code that will work correctly with procedures written in any high-level language:

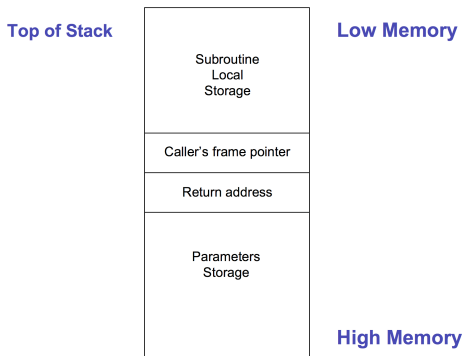


Figure: A standard stack frame

Frame Pointer

- The new pointer `fp` is relevant to the execution of a procedure.
- Using the stack pointer `sp` to `push()` and `pop()` elements, and to calculate the location of the parameters used by the procedure. This presents a problem, since the `sp` can change during the execution of a procedure
- Data on the stack frame cannot reliably be referenced through offsets from the `sp`
- The stack pointer can be used only immediately after the invocation of the procedure, since the `sp` is not guaranteed to have the same value throughout
- The `fp` is set to a fixed value within the stack frame, typically pointing just above the return address
- In the 68K `a7` is the `sp`, and by convention `a6` is typically used as the `fp`, although `a6` has no special properties.
- We can use the value in `a6` to calculate all the necessary offsets to run the procedure

Frame Pointer

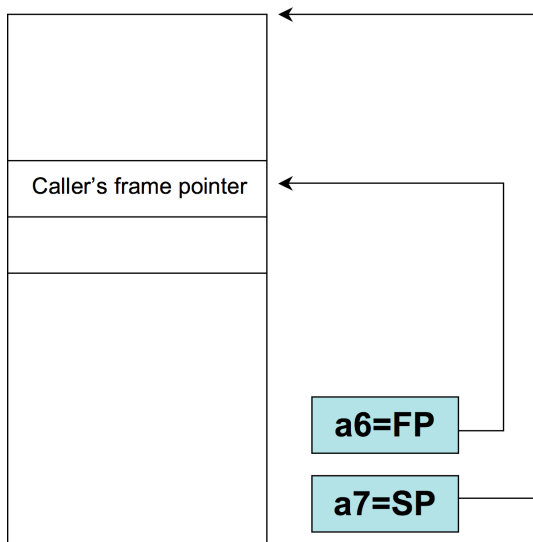


Figure: A standard stack frame and its pointers

Nested Calls

- A subroutine may call another one while executing
- The stack frame for the called routine is naturally built on top of the caller's
- We have to change the value of `a6` to point to the new invoked routine
- We store in the location pointed to by `a6` the address of the caller's `fp`

Nested Calls

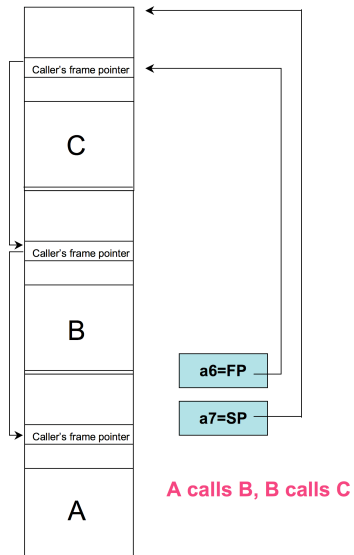


Figure: Nested stack frames

Nested Calls

- The *address* of the fp is used to calculate the offsets for the running routine
- The *contents* of that location is used to point to the caller's fp. This:
 - allows a called procedure to access values (variables, parameters, etc.) of the caller, and of the caller's caller, etc., by following the chain of pointers
 - since a caller is suspended while an invocation is proceeding, the caller cannot access the data in the invoked procedure
 - makes possible to implement an arbitrary number of such calls
- In particular, a routine may call itself repeatedly: this is called *recursion*

link

- The `link` instruction takes an argument register — assumed to be `a6` — and an immediate constant
- It is intended to be the first instruction executed by a called routine

`link an,d` ; `an` is assumed to be `a6`

- 1 Pushes `an` on the stack, that is, it copies `an` at `-(sp)`.
- 2 Puts `sp` into `an`, so the current stack pointer `sp` is pointed to by the frame pointer register, `an`.
- 3 Changes `sp` to `sp-d`, i.e. decrements the stack pointer `sp` by `d` (`d` is assumed to be the space necessary for local variables, see below).

link

This is exactly what is required by the call:

- just before a routine is called the parameters are pushed on the stack, thus building the lower part of the activation record
- the call to the routine pushes the return address on the stack, on top of the parameter space
- the link instruction pushes the current `an` on the stack, because this is pointing to the caller's activation record
- a space of size `d` is allocated on the stack to be used by local variables

unlink

- The instruction `unlink` is intended to be the last instruction executed by the routine, just before the `rts` instruction:

```
    unlink an      ; an is assumed to be the frame pointer
```

- The address register `an` is assumed to be the frame pointer
- The `unlink` instruction replaces the value of the stack pointer `sp` with the contents of `an`, and then pops the stack into `an` (i.e. loads into `an` the value $(sp)+$)

unlink

- The sequence is:
 - 1 $an \rightarrow sp$, change the value of the sp to the contents of an , so the stack pointer and the frame pointer have the same value, in effect deallocating the local variables space.
 - 2 $(sp)+ \rightarrow an$, put the value stored on the stack (the caller's frame pointer) into an and deallocate that stack space, so the stack pointer now points just above the return address ready for the rts instruction
- The routine has left the stack as it found it (the golden rule), only the parameters to the call remain on the stack. It is responsibility of the caller to remove these
- Return addresses and passed arguments are always positive relative to the frame pointer fp

Example 1

```
move.w    d0,-(sp)    ;push parameter #1 onto stack
move.w    d1,-(sp)    ;push parameter #2 onto stack
jsr       sbrt        ;jump to subroutine sbrt
```

```
sbrt      link        a6,-#$8    ;establish fp and local storage
          . . .
move.w    10(a6),d5    ;retrieve parameter #1 using fp
          . . .
unlink    a6           ;fp re-established
rts       ;deallocate stack frame and return
```

Example 2

```
n      equ      8           ;8 bytes for output
m      equ      8           ;8 bytes for local variables

      add.l      #-n,sp      ;put output area on stack
      move.l     arg,-(sp)    ;put argument on stack
      pea       x            ;address of data table on stack
      jsr       subr         ;goto subroutine
      add       #8,sp
      move.l     (sp)+,d1     ;read outputs
      move.l     (sp)+,d2
      . . .

subr   link      a1,#-m      ;save old sp
      . . .
      move.l     local1,-4(a1) ;save old variables
      move.l     local2,-8(a1)
      . . .
```

Example 2

```
. . .  
add.l    #1,-4(a1)        ;change a local variable  
movea.l  8(a1),a2         ;get x  
. . .  
move.l   output,16(a1)    ;push an output  
. . .  
unlk     a1  
rts
```

```
local1    dc.l    $98765432    ;local variables  
local2    dc.l    $87654321  
output     dc.l    'adcb'      ;output value
```

Example 3

```
; func(a, b); a and b parameters
; a6 is the frame pointer and a7 the stack pointer,

        move        a, -(a7)           ; push a on the stack, a7
        move        b, -(a7)           ; push b
jsr      func                ; push return address
                                ; and jump to func

; int func(int a, int b)
        link        a6,#-8             ; push a6, move a7 into a6
                                           ; a6 is now the frame pointer
                                           ; decrement a7 by 8
                                           ; to allocate a and b
                                           ; a6 is fp and is used to
                                           ; reference local variables
                                           ; a7 is sp and is used to
                                           ; store return addresses
                                           ; for function calls
```

Example 3

```
...  
    unlk a6                                ; undo link: copy a6 into a7,  
                                           ; retrieve old a6  
    rts                                   ; pop return address
```

Scope and Visibility

- A sequence of procedure calls results in a collection of stack frames piling up
- When the last procedure call is executing, all the other caller procedures have stopped executing.
- Since `procedure3()` stack frame is wiped out when it returns, `procedure2()` has no chance to be able to use any of the data on `procedure3()` stack frame.
- The stack frame for `procedure2()` is available when `procedure3()` is executing, and the data is accessible following the pointer from `procedure3()` to `procedure2()` frame pointer, so `procedure3()` is able to access data on `procedure2()` stack frame.
- The same applies for `procedure1()`, so `procedure3()` can access data in `procedure1()` stack frame.

Scope and Visibility

This characteristic relates to *variable scope and visibility*:

- The definition of a variable is restricted to the block where it has been defined
- Local variables of an inner block are 'invisible' to outer blocks
- A calling procedure is unable to access variables or other data local to the called procedure
- A called procedure is able to access variables or other data local to the calling procedure